Prof. Jürgen Sauer

Datenbanken

Skriptum zur Vorlesung im SS 2007

Inhaltsverzeichnis

1. TYPOLOGIE DER DATENBANKSYSTEME 7

1.1 Grenzen der herkömmlichen Datenverarbeitung 7

1.2 Erläuterung von Begriffen 10
1.2.1 Datenbanken und Datenbanksyteme 10
1.2.2 Informationssysteme 11
1.2.2.1 Dokumentations-Systeme 13
1.2.2.2 Data Warehouse 15
1.2.2.3 Expertensysteme 16
1.2.2.4 Data Mining 18
1.2.2.5 Informationssysteme 18
1.2.3 Klassifikation von Datenbanken 21
1.2.3.1 Klassifikationsmerkmale 21
1.2.3.2 Formatierte und unformatierte Datenbanken 21
1.2.3.3 Daten- und Speicherstrukturen 23
1.2.3.4 Grundfunktionen der Datenbanksoftware 26
1.2.3.5 Data Dictionary / Directory System (DD/D-System) 28

1.3 Datenbankmodelle für formatierte Datenbanken (DB) 31
1.3.1 Beschreibung der Daten in formatierten Datenbanken 31
1.3.1.1 Entitätsmengen und ihre Beziehungen 31
1.3.1.2 Beziehungen und Beziehungsattribute 34
1.3.2 Das relationale Datenbankmodell 35
1.3.2.1 Grundlagen 35
1.3.2.2 Normalisieren 36
1.3.3 Das Entity-Relationship Modell 39
1.3.4 Legacy-Datenbanken 51
1.3.4.1 Das netzwerkorientierte Datenbankmodell 51
1.3.4.2 Das hierarchische Datenbankmodell 56
1.3.5 Das Koexistenz-Modell 61
1.3.6 Grundkonzepte von Datenmodellen 66
1.3.7 Das Datenbankmodell für objektorientierte Datenbanken 67
1.3.7.1 Die Basis objektorientierter Datenbanknen 67
1.3.7.2 Grundlagen objektorientierter Systeme 69
1.3.7.3 Objektrelationale Datenbanken 72
1.3.7.3.1 Objektrelationale Konzepe 72
1.3.7.3.2 Objektrelationale Unterstützung in SQL3 74
1.3.8 Das UML-Modell 75
1.3.8.1 Die Diagramme 75
1.3.8.2 Schema-Modellierung 87
1.3.8.3 Datenbankmodellierung mit der UML 89
1.3.9 Semistrukturierte Datenmodelle 90
1.3.9.1 Das Object Exchange Model 90
1.3.9.2 XML-Datenmodell 91
1.3.9.2.1 XML-Dokumentenstruktur 92
1.3.9.2.2 XML-Standards 95
1.3.9.3 Abbildung von XML-Strukturen auf Datenbankstrukturen 98
1.3.9.4 Abbildung von XML-Daten auf Datenbanken (und umgekehrt) 100

1.4 Standards 103
1.4.1 CODASYL-Konzept 103
1.4.2 IMS 103
1.4.3 System R und SQL 105
1.4.3.1 System R 105

1.4.3.2 Standard-SQL (Structured Query Language) ... 105

1.5 Klassifikation der DB-Anwendungen ... 125
1.5.1 Elementare Anwendungsformen ... 125
1.5.2 Transaktionsbetrieb ... 128
1.5.2.1 Transaktionen ... 128
1.5.2.2 Das Zwei-Phasen-Commit-Protokoll ... 133
1.5.2.3 Transaktionsmonitor (Operating Systeme für Transaction Processing) ... 135
1.5.3 Client-Server-Systeme ... 136
1.5.3.1 Fernzugriff in Netzen aus autonomen Rechnern ... 136
1.5.3.2 Client-Server-Architekturen ... 141
1.5.3.2.1 Architekturformen ... 141
1.5.3.2.2 2-Tier und 3-Tier Client-/Server-Architekturen ... 147
1.5.3.3 Client/Server und Internet / Intranet ... 149
1.5.3.4 Web-Services ... 157
1.5.3.5 Oracle Application Server ... 158

1.6 Verteilte Systeme ... 159
1.6.1 Verteilte Datenbanken ... 159
1.6.2 Datenbankrechner und Datenbankmaschinen ... 167

1.7 Integritätsbedingungen und Integritätsregeln ... 167
1.7.1 Integritätsbedingungen ... 168
1.7.2 Integritätsregeln ... 169

2. RELATIONALE DATENBANKEN ... 171

2.1 Entwurf relationaler Datenbanken durch Normalisieren ... 171
2.1.1 Normalformen ... 171
2.1.1.1 Relationen in der 1. Normalform ... 171
2.1.1.2 Die zweite Normalform (ZNF) ... 173
2.1.1.3 Die dritte Normalform (DNF) ... 174
2.1.2 Spezielle Normalformen ... 175
2.1.3 Der Normalisierungsprozeß ... 178
2.1.3.1 Die Normalisierung als Zerlegungsprozeß ... 179
2.1.3.2 Der Syntheseprozess ... 182

2.2 Mathematische Grundlagen für Sprachen in relationalen Datenbanken ... 185
2.2.1.1 Die Basis: Mengenalgebra ... 185
2.2.1.2 Operationen der relationalen Algebra ... 186
2.2.1.3 Die Operationen der relationalen Algebra in Standard-SQL (SQL/89) ... 191
2.2.2 Das Realtionenkalkül ... 193
2.2.2.2 Prädikatenlogik ... 194
2.2.2.3 Logische Systeme: Prolog ... 199
2.2.2.4 Relationenkalkül und Prädikatenlogik ... 214
2.2.3 Datenbankmanipulationssprachen mit Bezugspunkten zur Relationenalgebra bzw. zum Relationenkalkül ... 215

2.3 SQL ... 216
2.3.1 SQL/92 ... 216
2.3.2 Oracle-SQL ... 226
2.3.2.1 Das Datenbanksystem Oracle ... 226
2.3.2.1.1 Speichern und Verwalten von Daten ... 226
2.3.2.1.2 Benutzer, Rollen und Berechtigungen ... 227
2.3.2.2 Aufbau der Datenbank ... 229
2.3.2.3 Kommunikation zwischen Benutzer und System über SQL*PLUS ... 231
2.3.2.4 SQL-Anweisungen in Oracle ... 232
2.3.2.5 Datenbankprogrammierung ... 253
2.3.2.5.1 PL/SQL ... 254
2.3.2.5.2 Constraints ... 267

2.3.2.5.3 Trigger ... 269
2.3.2.5.4 Stored Procedures, Functions, Packages ... 272
2.3.2.6 Datenbank-Verwaltung ... 274
2.3.3 Rekursive und iterative Abfragen mit SQL ... 277
2.3.4 Einbindung von SQL in prozedurale Sprachen ... 278
2.3.4.1 Embedded SQL ... 278
2.3.4.1.1 Embedded SQL in Oracle mit dem Precompiler Pro*C ... 283
2.3.4.1.2 Embedded SQL mit Java (SQLJ in Oracle) ... 290
2.3.4.2 Dynamisches SQL ... 297
2.3.4.3 Call-Schnittstelle (CLI) ... 299
2.3.4.3.1 ODBC ... 299
2.3.4.3.2 JDBC ... 302
2.3.4.3.3 ORACLE Call Interface (OCI) ... 317

3. OBJEKTRELATIONALE DATENBANKEN ... 323

3.1 Objektrelationales SQL in SQL:99 ... 323
3.1.1 Objektrelationales Typsystem ... 323
3.1.2 Datendefinition ... 323
3.1.3 Anfragen ... 324
3.1.4 Datenmanipulation ... 326

3.2 Objektrelationales SQL in Oracle ... 327
3.2.1 Objektrelationales Typsystem ... 327
3.2.1.1 Neue Basisdatentypen ... 327
3.2.1.2 Aggregations-Typen ... 331
3.2.1.2.1 Variabler Arraytyp ... 331
3.2.1.2.2 Tabellentyp und verschachtelte Tabellen (nested tables) ... 331
3.2.1.3 Abstrakte Datentypen ... 333
3.2.1.3.1 User Defined Types (Benutzerdefinierte Datentypen, Objekttypen) ... 333
3.2.1.3.2 User Defined Methods ... 334
3.2.1.4 Referenz-Typ ... 340
3.2.1.5 Objekttabellen und OIDs ... 340
3.2.1.6 Objekt-Views ... 345
3.2.2 Abfragen in Objektstrukturen ... 348
3.2.2.1 Abfragen von Objekttabellen und Spalten ... 348
3.2.2.2 Nützliche Funktionen für Objektinstanzen und Objekttabellen ... 349
3.2.2.3 Abfrage von Werten verschachtelter Objektinstanzen ... 349
3.2.2.4 Abfrage von Collection ... 349
3.3.1 Anwendungsprogrammierung in PL/SQL (Object PL/SQL) ... 350
3.3.2 Anwendungsprogrammierung mit Java ... 350
3.3.2.1 JDBC 2 ... 350
3.3.2.1.1 Collection Typen ... 350
3.3.2.1.2 CLOB, BLOB ... 350
3.3.2.1.3 Benutzerdefinierter Typ ... 350
3.3.2.1.3.1 Struct ... 350
3.3.2.1.3.2 Mit eigener Klasse ... 351
3.3.2.1.4 Referenz-Typ ... 351
3.3.2.2 SQLJ ... 351
3.3.2.3 JPublisher ... 351

4. XML-DATENBANKEN ... 353

4.1 Konzepte und Standardisierung ... 353
4.1.1 XML ... 353
4.1.1.1 XML-Dokumente ... 353
4.1.1.2 Bestandteile der Sprache und deren DTD-Definition ... 355
4.1.1.3 XML-Schema (Schemadefinitionssprachen) ... 359
4.1.2 Zusammenspiel von XML und Datenbanken ... 362

4.2 XML und Oracle 365
4.2 1 Speichern von XML-Daten in Oracle 365
4.2.2 Funktionsvorrat zur Unterstüzung von XML-Applikationen 366
4.2.3 XML-SQL Utility 367

EMPFOHLENE LITERATUR 369

ÜBUNGEN 371

AUFGABENBLÄTTER 373

1. Typologie der Datenbanksysteme

1.1 Grenzen der herkömmlichen Datenverarbeitung

Es heißt: Datenbanksysteme überwinden die Grenzen herkömmlicher Datenverarbeitung. Das kann sich nur auf das Bearbeiten von Dateien mit Dateisystemen beziehen. Das zeigt bspw. folgende 1. Aufgabe, die sehr vereinfacht eine maschinelle Gehaltsabrechnung beschreibt:

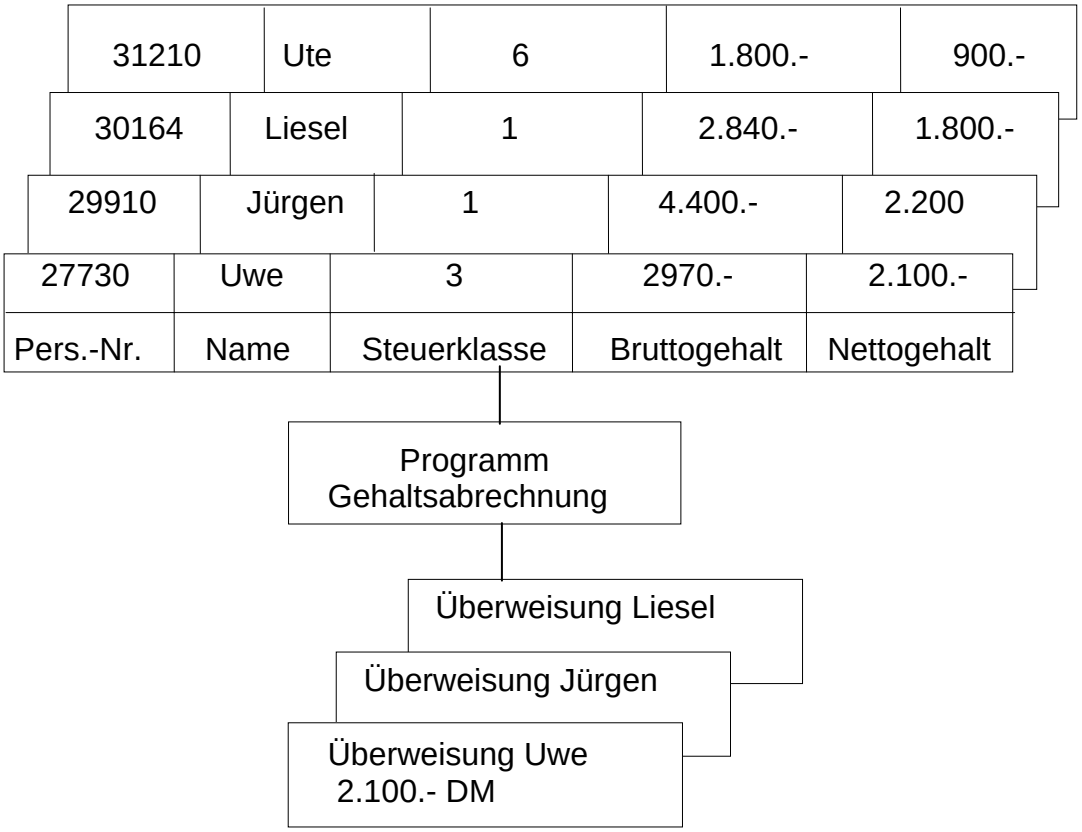


Abb. 1.1-1: Datenfluß zur 1.Aufgabe

Das Programm liest einen Datensatz in einen festen Bereich des Hauptspeichers ein und baut die Druckausgabe auf. Diese Druckausgabe besteht aus Überweisungen auf das Konto des Mitarbeiters bei seiner Bank.
Eine 2. Aufgabe ist: Beschriftung von Aufklebern mit den Adressen der Mitarbeiter (z.B. für den Versand von Mitteillungen).
Es gibt zwei Möglichkeiten, diesem Programm die Datenfelder "Name" und "Adresse" zur Verfügung zu stellen:

Lösung 1: Erweiterung des bestehenden Datensatzes für die Gehaltsabrechnung um ein Adressenfeld

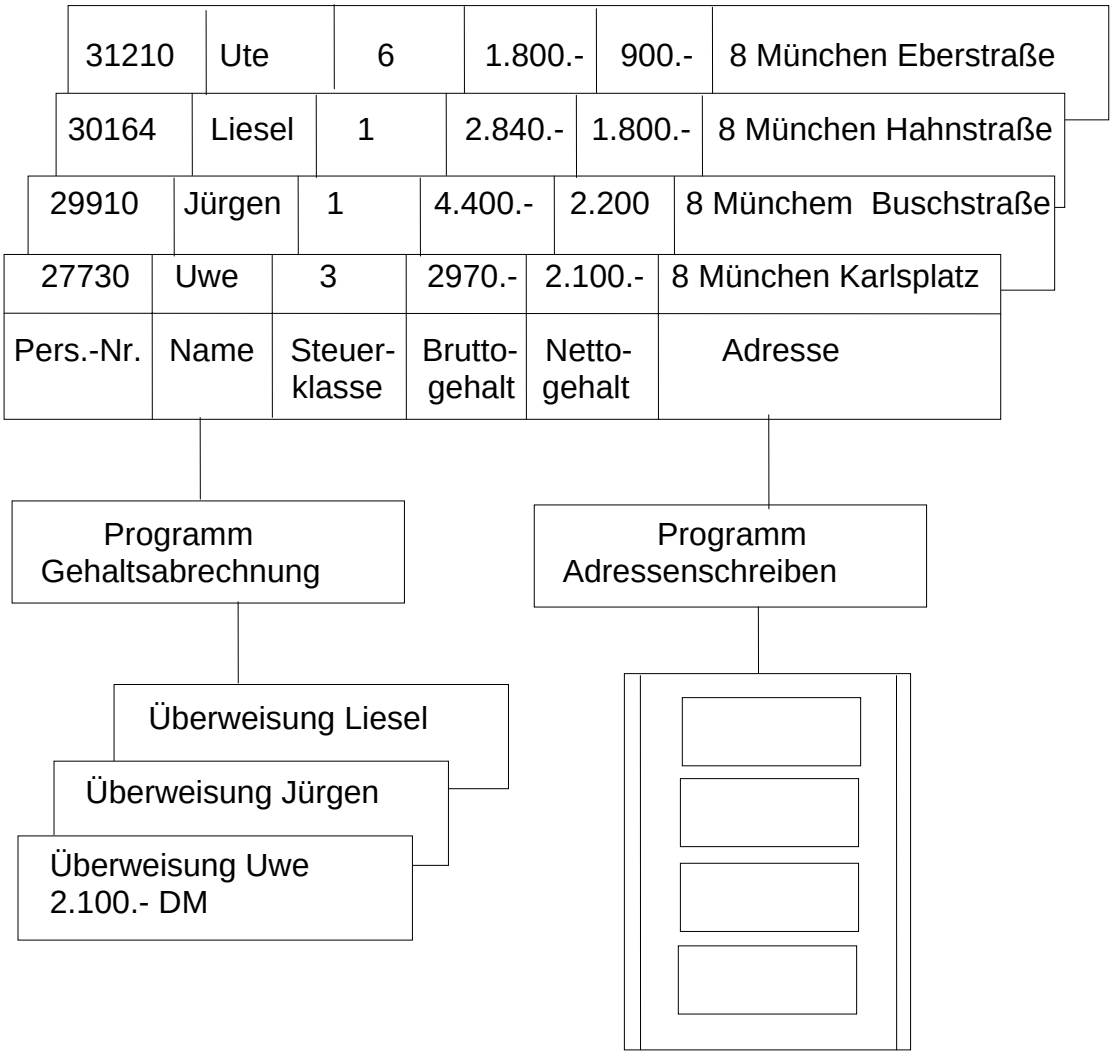


Abb. 1.1-2: Datenfluß zur 2. Aufgabe

Aus dieser Lösung ergeben sich zwei Schwierigkeiten:

1. Der Datensatz vergrössert sich. Das bestehende Programm Gehaltsabrechnung muß geändert werden, denn der Eingabebereich muß entsprechend vergrössert werden.Erweiterung von Datensätzen bedeutet: Änderung aller bestehenden Programme, die mit den Datensätzen arbeiten.
2. Das Programm, das die Aufkleber mit Adressen beschriftet, muß den erweiterten Datensatz einlesen. Damit kann man in diesem Programm auch alle anderen Personaldaten verarbeiten. Die Daten sind nicht mehr vertraulich (Datenschutz!).

Lösung 2:

- Der Datenbestand für die Gehaltsabrechnung bleibt unverändert. Damit entfallen Programmänderungen
- Der Datenbestand für die Adressenschreibung wird völlig neu aufgebaut



Abb. 1.1-3: Datenbestand für die Adressenschreibung

Es ergeben sich aus dieser Lösung allerdings zwei neue Probleme: Bestimmte Ordnungsbegriffe sind doppelt zu speichern (Pers.Nr., Name). Man spricht von Datenredundanz (gleiche Informationen sind mehrfach abgespeichert).
Daraus ergibt sich:

1. Zusätzlicher Bedarf an Speicherplatz auf externen Speichereinheiten
2. Unterschiedlicher Stand der Datenbestände, wenn nicht alle Änderungen in mehrfach gespeicherten Daten durchgeführt werden.

Eine 3. Aufgabe ist: Für jeden Mitarbeiter sind Informationen über seine Ausbildung bzw. Erfahrungen und die Möglichkeiten seiner Beschäftigung zu speichern. Mitarbeiter Müller kann z.B. im Unternehmen zwei Beschäftigungen ausüben. Für jede Beschäftigung hat er eine entsprechende Ausbildung bzw. Erfahrung.

Es ergeben sich wieder zwei Lösungen für die 3. Aufgabe:

Lösung 1: Erweiterung des gemeinsamen Datensatzes
Lösung 2: Aufbau eines dritten Datenbestandes

Probleme:

Wie viele Beschäftigungen kann ein Mitarbeiter möglicherweise dann ausüben?
Wie viele Ausbildungen bzw. Erfahrungen kann er haben?
Wieviel Speicherplatz muß für neue Ausbildungen bzw. Erfahrungen im Datensatz je Mitarbeiter frei gelassen werden?

Folgerungen

In der herkömmlichen Datenverarbeitung treten eine Reihe von Problemen auf. Diese Probleme verhindern den Aufbau von Informationssystemen. Folgende Forderungen sind daher von einem System zu erfüllen, das diese Probleme lösen soll:

1. Keine Programmänderung bei Erweiterung von Datensätzen
2. Zugriff nur auf solche Daten im Programm, die für die Problemlösung notwendig sind
3. Redundanzfreie Speicherung von Informationen
4. Beliebige Anzahl von Einfügungen solcher Informationen, die mehrfach auftreten können (z.B. mehrere Ausbildungen der Mitarbeiter)

1.2 Erläuterung von Begriffen

1.2.1 Datenbanken und Datenbanksyteme

Die Grenzen herkömmlicher (nicht integrierter) Datenverarbeitung (sog. Insellösungen) sind zu überwinden. Das kann durch eine zentrale Datenhaltung in einem Datenbanksystem erreicht werden. Datenbanken sind in der geschilderten Weise die Verwirklichung des Gedankens der integrierten Datenverarbeitung.

Was versteht man demnach unter einer **Datenbank** bzw. einem **Datenbank-system**?
Eine **Datenbank** (DB) ist eine Sammlung von Datenbeständen in einer hochgradig integrierten Speicherung (Speicher mit Direktzugriff) mit vielfältigen Verarbeitungsmöglichkeiten.
Ein **Datenbanksystem** (DBS) ist ein Software-Paket, das

- neutral große Datenmengen verwaltet
- die Beziehungen der möglichst redundanzfreien Daten zueinander kontrolliert
- unterschiedliche Verarbeitungs- und Auswertungsmöglichkeiten der Daten gestattet.

Das bedeutet: Alles, was oft als Datenbank (z.B. Oracle, Access, dBASE) angeboten wird, sind Datenbanksysteme (**DBS**). Das sind Programme, mit deren Hilfe Datenbanken für interne Zwecke aufgebaut und gepflegt werden können. Es handelt sich dabei um Datenbankverwaltungs-Systeme (Datenbankmanagement-Systeme, DBMS), die nur die Organisation, nicht aber den Inhalt der Datenbank bestimmen.
Streng genommen bezeichnet der Begiff „Datenbank“ eine nach bestimmten Regeln aufgebaute Datensammlung (Texte, Tabellen bzw. Zahlen, Zeichen, Graphiken). Auf diese Daten besteht eine Zugriffsmöglichkeit, die über vielfältige Kombinationsmöglichkeiten des Suchbegriffs gewünschte Informationen bereitstellt. So wird heute weltweit der Zugriff auf mehr als 6000 öffentlich zugängliche Datenbanken angeboten. Sie werden von zahlreichen Instituten, Zeitschriften, Zeitungen, Nachrichtenagenturen, Vorlagen und amtlichen Stellen gefüllt. In **externen Datenbanken** arbeitet der Anbieter einer Datenbank mit Hilfe von Datenbanksystemen zusammen. Über Datenbanksysteme werden externe Quellen (Zeitschriften, Börsenmitteilungen, Gesetze, Urteile, etc.) ausgewertet und in eine für die Datenbank geeignete Form gebracht. Im Gegensatz dazu enthalten **interne Datenbanken** nur Daten, die im eigenen Umfeld des Datenbankadministrators anfallen.

1.2.2 Informationssysteme

Elemente betrieblicher Informationssysteme

Die kleinste Informationseinheit ist das Zeichen (Element aus einer Darstellung von Information mit verschiedenen Elementen, dem Zeichenvorrat). Im Normalfall reicht ein einzelnes Zeichen nicht zur Darstellung von Information, man benötigt eine Zeichenfolge. Wird eine Zeichenfolge übertragen, spricht man von einer **Nachricht**. Eine Nachricht wird zur Information, wenn sie für den Empfänger eine Bedeutung hat.
Eine Zeichenfolge folgt den Regeln einer Grammatik: mehrere Zeichen formen sich zu Worten, Worte formen sich zu Sätzen. Diese linguistischen Einheiten (Zeichen bis Sätze) sind Platzhalter für Objekte, auf der sich der Sender einer Nachricht bezieht. Die Platzhalter werden zur Information, wenn die übermittelte Zeichenfolge aus einem Kontext in Beziehung gesetzt wird.
Ein Informationssystem (**information system**, **IS**)[1] besteht aus Menschen und Maschinen, die Information erzeugen und/oder benutzen und die durch Kommunikationsbeziehungen untereinander verbunden sind.
Ein **rechnergestütztes Informationssystem** (compter based information system) ist ein System bei dem Erfassung, Speicherung, Übertragung und/oder Transformation von Information durch den Einsatz der Informationstechnik teilweise automatisiert ist.

Was gehört alles zu einem **Informationssystem**?

Eine Datenbank[2] liefert **Fakten** für ein Informationssystem. Ausgangspunkt solcher Systeme ist der Benutzer. Er stellt Fragen, das Informationssystem gibt Antworten. Einfache Anfragen können direkt aus der Datenbank beantwortet werden.

Bsp.:

Datenbank	Frage
Buchhaltung	Letzjähriger Gesamtumsatz
Lexikon	Geburtsjahr von J. S. Bach

Für komplizierte Fragen sind zusätzlich Methoden zur Kombination bestimmter Daten aus der Datenbank nötig
Methodenbanken sind Programme für mathematische Verfahren (Matrizen-, Differential- und Integralrechnung), statistische Auswertungen und Operations Research. Methodenbanken sind auf Großrechern (Mainframe) immer noch selten. Häufig anzutreffen dagegen sind integrierte Programmpakete für Mikrocomputer, die Programme zur Datenbankanwendung sowie für Präsentationsgrafik, Tabellen-kalkulation und Textverarbeitung enthalten.
Damit ist die Basis geschaffen für weitere Nachforschungen, die mehr als Faktenermittlung und zugehörige spezifische Auswertung umfassen.

[1] Vgl. Hansen, Hans Robert und Neumann, Gustaf: Wirtschaftsinformatik I, Lucius&Lucius, Stuttgart, 9. Auflage 2005
[2] vgl. 1.2.1

Bsp.:

Methodenbank	Datenbank	Frage
Betriebsabrechnung	Buchhaltung	Vollkosten der Maschine X je Stunde
Prognose-Methoden	Bevölkerungsdaten	Schulanfänger in 10 Jahren

Manchmal sind weitere Nachforschungen nötig.

Bsp.:

Datenbank	zusätzliche Nachforschungen	Frage
Buchhaltung	Geschäftsberichte der Konkurrenz	Personalkosten im Vergleich zur Branche

In speziellen Anwendungen muß der Datenbestand für den Fragesteller durch eine „Organisation der Datenverarbeitung“ aufbereitet werden. Methoden, Recherchen (Nachforschungen) und Speicherung der Daten bilden durch diese „Organisation der Datenverarbeitung“ ein Informationssystem.

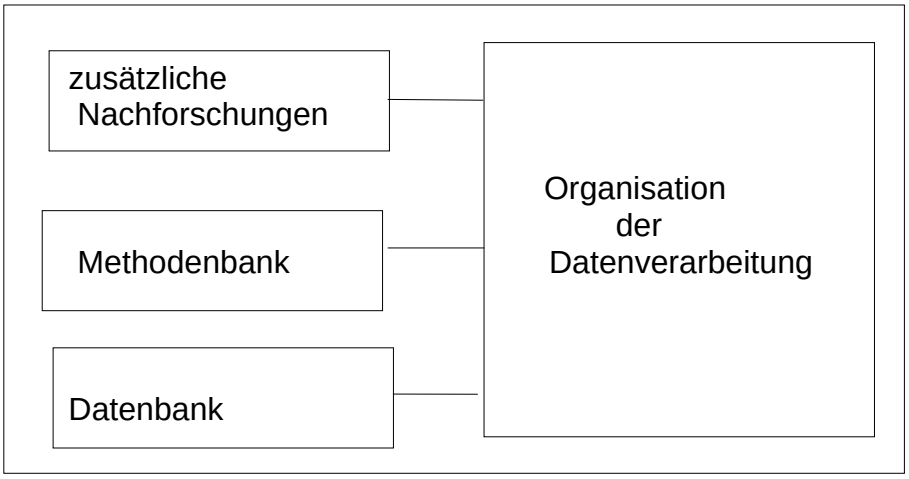


Abb. 1.2-1: Aufbau eines Informationssystems

Die „Organisation der Datenverarbeitung“ ist weitgehend bestimmt durch die Anwendung. Solche Organisationsformen haben sich herausgebildet für Dokumentationssysteme, Expertensysteme, „Data Warehouse“-Systeme.

1.2.2.1 Dokumentations-Systeme

Informations- bzw. Dokumentations-Systeme enthalten:

Stichwortkataloge

Die Titel oder die Zusammenfassungen der zu dokumentierenden Zeitschriftenartikel oder Bücher werden auf aussagenkräftige Begriffe durchsucht. Weggelassen werden dabei alle Wörter, die nichts zum Suchprozeß beitragen (Artikel, Präpositionen, Konjunktionen, wenig aussagekräftige Wörter).

Schlagwortkataloge

Nicht aus dem ursprünglichen Dokument bestimmte Stichwörter sondern speziell ausgewählte Schlagwörter aus einem Schlagwortregister bilden hier die Grundlage. Der Benutzer muß seine Abfrage mit diesen Schlagwörtern (Deskriptoren) aufbauen (vgl. Schlagwortverzeichnis der Association of Computing Machinery (ACM)).
Deskriptoren können aber zu Problemen der folgenden Art führen:

- das Synonymenproblem (Mehrfachbenennung eines Begriffs)
- das Polysemenproblem (Verwendung einer Begriffsbenennung für verschiedene Begriffe)
- das Äquivalenzproblem

Verwandte Deskriptoren werden zweckmäßigerweise zu Äquivalenzklassen zusammengefaßt. Damit kann eine Dokumentensuche u.a.a. über einen zunächst gegebenen Deskriptor erweitert werden. Das Problem dabei ist, inwieweit die zu einer Äquivalenzklasse zusammengefaßten Deskriptoren wirklich "gleich" sind.

Thesaurus

Ein **Thesaurus** (Wortschatz) ist ein Verzeichnis von verschiedenen Schlagwörtern.
Thesauri werden verwendet:

- als vordefinierte Deskriptorenliste
- als Synonymenwörterbuch
- zur Gruppierung, Klassifizierung oder Strukturierung von Deskriptoren

In verschiedenen Fachgebieten werden umfassende Thesauri angeboten.

Bsp.: "Engineers Joint Council Thesaurus"
Ein Auszug aus diesem Thesaurus enthält Vorzugsbenennungen und zusätzliche Begriffsbestimmungen, die in Beziehung zu den Vorzugsbenennungen gesetzt sind. Die Beziehungen sind gekennzeichnet durch:

UF (used for)
Die vorstehende Vorzugsbenennung ist synonym mit der durch UF gekenn-zeichneten Begriffsbenennung.
BT (broader term)
Die mit BF gekennzeichnete Begriffsbenennung umfaßt die vorstehende Vor-zugsbenennung in ihrer Bedeutung.
NT (narrower term)
Die mit NT gekennzeichnete Begriffsbenennung ist hins. ihrer Bedeutung in der vorstehenden Begriffsbenennung enthalten.

USE
Die mit USE gekennzeichnete Begriffsbenennung ist als synonyme Vor-zugsbenennung für die vorstehende Begriffsbenennung zu verwenden.
RT (related term)
Die mit RT gekennzeichnete Begriffsbenennung ist in ihrer Bedeutung mit der vorstehenden Vorzugsbenennung verwandt (, jedoch nicht im Sinne einer BT- oder NT-Beziehung).

Vorzugsbenennungen sind unterstrichen, z.B.:

```
Koaxialkabel
     UF Koaxialleitung
     NT Flüssigkeitsgefüllte Koaxialkabel
     BT Übertragungskabel
     RT Starkstromkabel

Koaxialleitung
     USE Koaxialkabel

Koaxialfilter
     BT Elektr. Filter
     RT Mikrowellenfilter
Kobalt
     BT Metall
```

Ein **Thesaurus** ist eine Dokumentationssprache[3], die aus Deskriptoren zur eindeutigen Begriffsbenennung (Vorzugsbenennung) und zusätzlichen Wörtern der natürlichen Sprache (ergänzende Begiffsbenennung, Hilfsmittel zur Darstellung von Beziehungen zwischen diesen Benennungen) besteht. Vorzugsbenennungen sind für die Informationswiedergewinnung die entscheidenden Größen. Unter ihnen gibt es keine Polyseme oder Synonyme. Vorzugsbenennungen sind in der Regel durch zusätzliche Begriffsbestimmungen ergänzt, damit eine vielschichtige Beschreibung der Dokumente möglich ist. Eine weitere zentrale Aufgabe in einem Thesaurus ist neben der Bestimmung von Vorzugsbenennungen und der darüber hinaus erlaubten Begriffsbenennungen die Festlegung von Beziehungen zwischen den Begriffsbenennungen.
Folgende Beziehungen findet man in Thesauri am häufigsten: „übergeordneter Begriff (OB), untergeordneter Begriff (UB), verwandter Begriff (VB)“. Wenn dann A bspw. Oberbegriff von B ist, ist zugleich B Unterbegriff von A. Wenn A mit B verwandt ist, ist auch B mit A verwandt. Die Beziehung (Relation) „verwandter Begriff“ ist damit symmetrisch.

Systematischer Katalog

Darunter verstehen Bibliothekare die Verwendung eines künstlichen Schlagwortverzeichnisses, nämlich der "universellen Dezimal-Klassifikation (UDK oder DK)".

Dewey's Dezimalklassifikation
Um 1870 entwickelte der amerikanische Bibliothekar Melvil Dewey ein Ordnungssystem für eine einheitliche Bücheraufstellung. Er teilte das gesamte "menschl. Wissen" in 10 Hauptabteilungen:

[3] Regeln für die Erstellung und Weiterentwicklung von Thesauri sind in DIN 1463 festgelegt

```
0  Allgemeine Medizin
1  Philosophie
2  Religion, Theologie
3  Sozialwissenschaften, Recht, Verwaltung
4  Sprachen
5  Mathematik
6  Technik, Medizin
7  Kunst
8  Literatur
9  Geschichte
```

Jeder der Hauptabteilungen teilt sich in 10 Abteilungen (00, .. ,09, 10, .. ,19,20 ...) und bei Bedarf jede dieser Abteilungen in 10 Unterabteilungen (000, 001, ...). Dieser Klassifikationsvorschlag fand eine sehr starke Verbreitung. Die verwendete Dokumentationssprache ist einfach zu lernen, denn sie besteht ja nur aus etwa 1000 Deskriptoren. Die Deskriptoren sind hierarchisch geordnet.
Die internationale Dezimalklassifikation
Das Klassifikationssystem von M. Dewey wurde bis heute immer wieder weiter überarbeitet. Durch weitere Untergliederung entstanden bereits bis 1973 70000 Deskriptoren. Die internationale Dezimalklassifikation wird heute gepflegt von der "Federation Internationale de Dokumentation". Diese Kataloge bilden die Hilfsorganisation für Suchfragen. Sie erlauben dem Dokumentationsbenutzer einen schnelleren Zugang zur gesuchten Information.

1.2.2.2 Data Warehouse

Ein Data Warehouse[4] ist eine Datenbank mit historischen Daten, die aus unterschiedlichen Datenquellen eines Unternehmens in eine seperate Datenbank kopiert werden. Herkömmliche Datenbanken (operative Systeme) bearbeiten das Tagesgeschäft. Informationen über z.B. die Umsatzsituation, Soll/Ist-Werte einer Budget-Planung sind daraus leicht abzuleiten. Informationen über die Umsatzentwicklung der letzten 6 Jahre, Soll/Ist-Entwicklung im Vergleich zum Vorjahr können nicht direkt ermittelt werden.Ein **Data Warehouse** stellt Datenmaterial bereit, derartige komplexe Fragen zu beantworten. Es enthält Daten (in unterschiedlicher Verdichtung), die für die Entscheidungsunterstützung der Benutzer relevant sind. Direktzugriff wird Endbenutzern durch einen Informationskatalog ermöglicht, der über Inhalte, Formate und Auswertungsmöglichkeiten des Data Warehouse Auskunft gibt.
Die Auswertung dieser Datenbank durch eine große Anzahl von Benutzern stellt große Anforderungen. Der Erfinder des Datenbankmodells für relationale Datenbanken, E.F.Codd, hat folgende Merkmale genannt, die Systeme für das "`Online Analytical Processing (OLAP)`" erfüllen sollen:

- Mehrdimensionale, konzeptionelle Sicht auf die Daten
- Transparenz und Integration in die operativen Systeme
- Zugänglichkeit unterschiedlicher Datenbanken über eine logische Gesamtsicht
- stabile, volumenunabhängige Antwortzeiten
- Client-/Server-Architektur
- Mehrbenutzerunterstützung

[4] Die Hersteller (z.B. Oracle, Sybase, Software A.G., IBM, Siemens etc.) vermarkten unter diesem Schlagwort alle möglichen Werkzeuge, die mit der Verwaltung, Integration und Auswertung großer Datenbestände zu tun haben.

- flexibles Berichtswesen
- unbeschränkt dimensionsübergreifende Operatoren

1.2.2.3 Expertensysteme

Ein Expertensystem ist ein Programm, das sich in irgendeinem Anwendungsbereich wie ein Experte verhält. Expertensysteme müssen fähig sein, Probleme zu lösen, die Expertenwissen in einem bestimmten Bereich verlangen. Sie sollen in irgendeiner Form Wissen verarbeiten (wissensbasierte Systeme).
Ein voll ausgebautes Expertensystem besteht im allg. aus 5 verschiedenen Komponenten:

- Die **Wissensbasis**
  bildet die Grundlage. Sie enthält die Kenntnisse des Experten, meistens in Form von Fakten und Regeln oder auch als Rahmen (Beschreibung von Objekten) und Skripten (Beschreibung von Abläufen).
- Die **Inferenz-Maschine** (inference machine)
  dient der Wissensauswertung. Sie sucht und verknüpft Fakten und Regeln nach einer vorgegebenen Strategie und produziert Forderungen und Ergebnisse
- Die **Erklärungskomponente** (explanation component)
  kann dem Anwender begründen, durch welche Regeln und Fakten ein Ergebnis zustande kam. Sie gibt dem Experten die Möglichkeit zu überprüfen, ob das System seine Schlußfolgerungen korrekt nachbildet
- Der **Dialogteil** (dialog management)
  führt das Gespräch zwischen Anwender und Rechner
- Die **Wissensadministration**
  ermöglicht dem System zu lernen, d.h. neues Wissen in die Wissensbasis einzufügen oder altes Wissen zu verändern, ohne daß dies explizit programmiert werden muß.

Was unterscheidet ein Expertensytem von herkömmlichen Datenbanken?
Aus den bisher vorliegenden Angaben könnte abgeleitet werden, es handle sich bei einem Experten-System um nicht viel mehr als eine Datenbank mit einem komfortablen Abfragesystem. Ein Expertensystem ist aber mehr. Drei Eigenschaften, die ein Datenbanksystem nicht besitzt, charakterisieren ein Expertensystem. Es ist heuristisch, lernfähig und selbsterklärend .
Datenbanken enthalten Fakten über die reale Anwendungswelt. Nur eine kleine Anzahl von Regeln kann in Datenbanksystemen, z.B. in der Form von Integritätsbedingungen, enthalten sein. Generell ist keine Speicherung von Regeln vorgesehen. So kann bspw. die Regel „Wenn ein Student Informatik studiert, dann mag er Prolog“ in konventioneller Datenbanktechnik nicht explizit gespeichert werden. Wissensbasen erlauben im Gegensatz zu Datenbanken die explizite Darstellung regelbasierter Informationen, aus denen Schlußfolgerungen (Ableitung von Informationen) gezogen werden können.

Verfahren zur Repräsentation von Wissen können eingeteilt werden in:

1. Logische Vefahren
   Die Wissensbasis wird durch Ausdrücke der formalen Logik präsentiert. Inferenzregeln wenden dieses Wissen auf die Lösung spezifischer Probleme an. Das am häufigsten verwendete Darstellungsschema ist das **Prädikatenkalkül** 1. Ordnung. Die Programmiersprache **Prolog** stützt sich auf dieses Kalkül und ist daher für die Implementierung von Wissensbasen mit logischen Repräsentationsverfahren besonders geeignet.
2. Netzwerk-Verfahren

Sie präsentieren das Wissen durch einen Graphen, bei dem die Knoten die Objekte und Konzepte des Problemgebiets darstellen, und die Kanten die Beziehung zwischen diesen Objekten oder Objekttypen. **Semantische Netze** sind dafür ein Beispiel. Im Gegensatz zu Datenbankmodellen (Trennung von Schema und Instanz) gehören Objekte (Instanzen) zur Repräsentation von Wissen hinzu. Netzwerkbezogene Verfahren sind demnach eine objektbezogene, graphische Darstellung von Wissen.

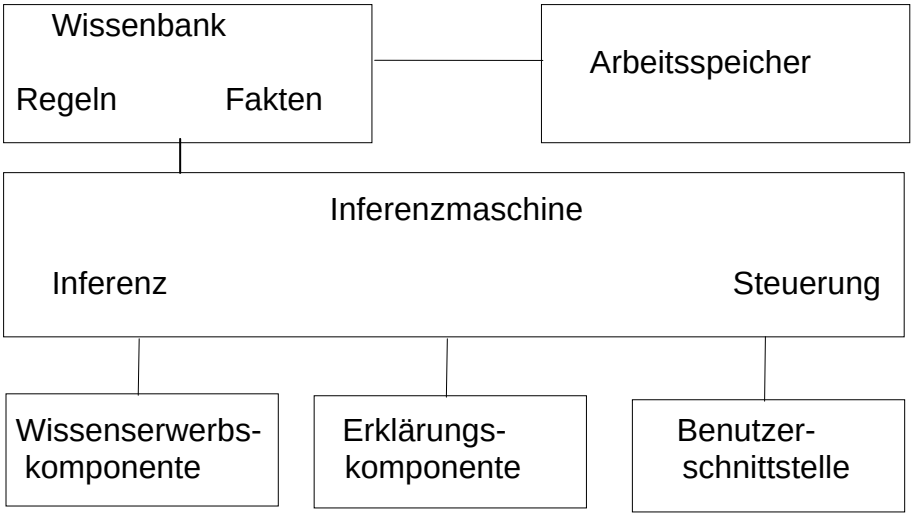


Abb. 1.2-2: Aufbau eines Expertensystems

Datenbanksysteme werden bzw. wurden für Anwendungen entwickelt, die sich folgendermaßen charakterisieren lassen:

- Die Daten weisen eine feste, vordefinierte Struktur auf
- Jeder Datensatz (jedes Tupel) beschreibt i.a. ein bestimmtes Objekt der Anwen-dung (ein Konto, ein Lagerbestand)
- Die Menge der über ein Objekt gespeicherten Daten ist i.a. klein (wenige 100 Bytes)
- Daten können mit hoher Frequenz eingefügt, modifiziert und gelöscht werden
- Es sind i.a. viele Benutzer gleichzeitig auf derselben Datenbank aktiv
- Die Daten müsen gegen Bedienungsfehler, Systemausfälle, Datenträgerverluste u.a. wirksam geschützt werden und automatisch wiederherstellbar sein

Expertensysteme wurden bzw. werden für Anwendungen konzipiert, die sich etwa durch folgende Punkte beschreiben lassen:

- Über einen bestimmten Wirklichkeitsausschnitt liegen eine Reihe feststehender Fakten und Regeln vor, sowie Gesetzmässigkeiten, Konventionen, Erfahrungen
- Das System soll dem Benutzer bei allgemeinen Problemlösungsaufgaben in der Anwendungsumgebung unterstützen, d.h.: Überprüfen von Entscheidungs-prozessen; Nachweis der Gültigkeit von Fakten; Untersuchen der Korrektheit von Aussagen
- Die hierzu benötigten Operationen sind: Auffinden einschlägiger Regeln für eine zutreffende Entscheidung; Ableitung neuer Fakten und Regeln; Bewertung von Entscheidungsvarianten
- Dem Benutzer muß eine Schnittstelle angeboten werden, an der in möglichst einheitlicher Weise die Struktur der momentanen Fakten und Regeln dargestellt und verändert werden kann.

Neben diesen anwendungsspezifischen Eigenschaften machen die meisten der heute verfügbaren Expertensysteme noch folgende Annahmen:

- Die Menge der zu verwaltenden Fakten und Regeln ist klein (klein genug, um vollständig im Hauptspeicher gehalten zu werden)

- Es arbeitet jeweils nur ein Benutzer auf derselben Fakten- und Regelmenge
- Die Sicherung und Wiederherstellung der Daten wird durch die Betriebsumgebung geleistet
- Änderung der Fakten und Regeln sind sehr selten

1.2.2.4 Data Mining

Fortschritte in Computer- und Datenbanktechnologie ermöglichen immer umfangreiches Sammeln von Daten mit der Problematik: "In riesigen Datenhalden "Informationsjuwelen“ finden". Ein Lösungsansatz des Problems ist: "Wissensentdeckung in Datenbanken (Knowledge Discovery in Databases (KDD)) mit der Entdeckungsmethode Data Mining.
Data Mining ist: "Die Anwendung verschiedener Analyse und Lernmethoden zur Entdeckung von Wissen in Datenbanken. (automatische Gewinnung prognostischer Information aus großen Datenbanken).

1.2.2.5 Informationssysteme

Electronic Data Interchange (EDI)

Ein zwischenbetriebliches IS vebindet Informationssysteme zweier oder mehrerer Betriebe. Das Spektrum der Zusammenarbeit umfaßt:

- elektronischer Austausch von Bestellungen (electronic data interchange, **EDI**)
- virtuelle Organisation
- Informationssysteme einer großen Anzahl von Betrieben verschiedener Branchen

Ein **EDI-System** ist ein System zur Abwicklung von Geschäftstransaktionen, bei denen ein Betrieb mit einem oder mehreren Geschäftspartnern (Lieferanten, Geschäftskunden, Behörden, usw.) entweder direkt oder über eine sog. Clearung-Stelle elektronische Geschäftsdokumente austauscht.
EDI einheitliche Normen (z.B. EDIFACT-Normen) bestimmen Inhalt und Sysntax von elektronisch zu übertragenden Daten. Eine Schlüsselrolle spielt hierbei die Verwendung der Extensible Markup Language (**XML**). Sie dient zur

- Grundlage zur Beschreibung von EDI-Dokumenten
- Einfachen und effizienten Weiterverarbeitung der Daten

XML

XML ist ein vom World Wide Web Consortium (W3C) entwickelter Standard, eine Metasprache, mit der man Auszeichnungen (Markups) zu Dokumenten für die Strukturbeschreibung hinzufügen kann. XML ermöglicht es, als ein für den Informationsaustausch nutzbares Format, interne Unternehmensdaten, über Systemgrenzen und Plattformen hinweg, im Internet nutzbar zu machen

```
<lexikon>
 <eintrag stichwort="Information">
   <herkunft>lat.</herkunft>
   <erklaerung num="1">Auskunft, Nachricht, Mitteilung, Belehrung</erklaerung>
   <erklaerung num="2"><siehe_auch>Fachinformation</siehe_auch></erklaerung>
```

```
  <erklaerung num="3"><anwendung>Informatik:</anwendung> die formulierte
    Unterrichtung nicht nur von Menschen, sondern auch von
    Organisationen und techn. Einrichtungen über Sachverhalte,
    Ereignisse, Abläufe. Die <siehe_auch>Informationstheorie
   </siehe_auch> versteht unter Informatik ein Maß, das den Zeichen einer
   Nachricht zugeordnet ist.
  </erklaerung>
  ...
 </eintrag>
</lexikon>
```

Abb.: XML-Dokumment zur Darstellung semistrukturierter Informationen (Quelle der Inhalte: Der Brockhaus in fünf Bänden)

Frei wählbare Tags ermöglichen die semantische Beschreibung der Daten zwischen einem öffnenden (Start-) und einem schließenden (Ende-) Tag. Diese semantische Beschreibung ermöglicht Dateninterpretation vom Menschen und von Maschinen (bzw. Programmen).

Informationssysteme aus XML-Dokumenten und Datenbanken

Informationssysteme enthalten Daten aus Textdokumenten und HTML-Dokumenten des World Wide Web. Text- und HTML-Dokumente sind in der Regel nicht so stark strukturiert wie Datenbankdaten. Sie besitzen jedoch eine interne, oft wechselnde und nicht streng typisierte Struktur. Solche Daten bzw. Dokumente bezeichnet man als semistrukturiert.

Bsp.:

```
<book>
  <author>Neil Bradley</author>
  <title>XML companion</title>
  <isbn>1-234-56789-0</isbn>
  <content>
    XML builds on the principles of two existing
    languages, <emph>HTML</emph> and ..
  </content>
</book>
```

Der wichtigste Faktor bei Auswahl einer Datenbank für den Einsatz von XML ist die Frage, ob man die Datenbank zur Speicherung von Daten oder Dokumenten benutzt. Man unterscheidet zwischen datenorientierten Dokumenten („**data centric**“), deren Schwerpunkt auf den Inhalten der einzelnen Strukturen liegt und dokumentorientierten Dokumenten („**document centric**“), bei dem das Dokument als ganzes betrachtet wird.

datenzentriert	
- Daten stehen im Mittelpunkt - einheitliche, strikte Struktur - SQL-ähnliche Suche erwünscht - Elementereihenfolge oft unwichtig - z.B. Katalogdaten, Rechnungen	<Rechnung> <Kunde> Meier </Kunde> <Artikel> Seife </Artikel> <Preis> 2,40 </Preis> </Rechnung>

Datenorientiertes XML ist durch eine relativ regelmäßige Struktur gekennzeichnet. Derartige Strukturen wurden normalerweise für maschinelle Verarbeitung entworfen.

Bsp.:

```
<order>
  <customer>Meyer</customer>
  <position>
    <isbn>1-234-56789-0</isbn>
    <number>2</number>
    <price currency=„Euro“>30.00</price>
  </position>
</order>
```

dokumentenzentriert	
- Daten und Struktur sind relevant - uneinheitliche, schwache Struktur - Volltextsuche erwünscht - Elementereihenfolge oft wichtig - z.B. HTML-Seiten, Buchkapitel	<Heading>Liedtext</Heading> Sie müssen erst den <bold> Nippel</bold> durch die Lasche zieh‘n, und mit der kleinen ...

Ein dokumenentenzentriertes Datenformat (Austauschformat, engl. document centered data format, exchange format) ermöglicht die gemeinsame Übertragung und Speicherung semantisch zusammengehörender Daten in einem einzelnen Dokument. Das Dokument wird in einem Format erstellt, das flexibel genug ist, um Strukturierung, Speicherung, Übertragung und (teil-) automatisierte Weiterverarbeitung beliebiger Dokumentinhalte zu ermöglichen. Man spricht hierbei von einem Dokument, das nicht nur für den Rechner geeignet ist, sondern auch vom Menschen gelesen werden kann.
Bsp.:

```
<content>
XML builds on the principles of two existing
 languages, <emph>HTML</emph> and
<emph>SGML</emph> to create a simple
mechanism ..
The generalized markup concept ..
</content>
```

semistrukturiert	
Mischform aus daten- und dokumentzentriert	<Song> <Titel> Der Nippel </Titel> <Interpret> Mike Krüger </Interpret> <Text>Sie müssen erst den <boldf> Nippel</boldf>…

Für die datenorientierte XML-Struktur kann man sich eine passende Tabelle in einer Datenbank vorstellen.
Bei dokumentorienter XML-Struktur könnte man sich eher die Speicherung vom XML-Dokument als ganzes in der Datenbank vorstellen.

XML-Datenbanken

XML-Daten in Datenbanken bilden eine geeignete Kombination von XML als Austauschformat für Daten unterschiedlicher Herkunft und den Vorteilen eines DBMS (Zugriff auf die Vielzahl der Informationen, die in der DB gespeichert sind). XML-Dokumente können wegen ihrer speziellen Aufzeichnungsysntax gezielt nach Informationen durchsucht werden. Eigene für XML entwickelte Abfragesprachen existieren bereits. Große Datenmengen in XML können durch Zerlegung in ihre

Elemente auf den Strukturen einer Datenbank zum schnellen Durchsuchen abgebildet werden.
Als XML-Datenbank werden Datenbanken oder Datenbanksysteme bezeichnet, die Daten im XML-Format speichern oder anderweitig mit XML-Daten umgehen können. Man kann daher XML-Datenbanksysteme in zwei Kategorien einteilen:

- XML enabled. Herkömmliche Datenbanksysteme, die ein Mapping auf ider ins XML-Format erlauben. Man bezeichnet die Vorgehensweise als datenorientiert.
- Native XML-Datenbanksysteme. Diese Systeme speichern Information, ähnlich wie bei der Speicherung von XML-Dokumenten in Dateien-Systemen, direkt als Dokumente ab. Sie werden auch als dokumentenorientiert bezeichnet.

1.2.3 Klassifikation von Datenbanken

1.2.3.1 Klassifikationsmerkmale

Es gibt in der Datenverarbeitung zwei Arten von Informationen, die wegen ihrer grundsätzlich verschiedenen Struktur auch bei Datenbanken unterschiedliche Behandlung erfordern:

1. Texte (schwach strukturiert)

Jede Zeichenkette (Wort) besitzt eine Bedeutung. Die Stellung der Worte im Text ergibt den logischen Zusammenhang

2. Daten (stark strukturiert)

Die Position der Daten innerhalb einer Zeichenkette weist dem Wert eine Bedeutung zu. Dadurch ist die Reihenfoge streng festgelegt.

Bsp.: Zahlenfolge '345670076'
Die Zahlenfolge sagt nichts aus, solange unbekannt ist:

- die ersten 2 Stellen (34) beinhalten den Schlüssel für eine Automarke
- die nächste Stelle (5) den Fabrikationsort
- die nächsten 4 Stellen (6700) den Autotyp

1.2.3.2 Formatierte und unformatierte Datenbanken

Formatierte Datenbestände

Die Abspeicherung der Daten erfolgt nach einem festen Schema (Format). Der Zugriff zu den Daten ist über Ordnungskriterien und Feldnamen gegeben.

Bsp.:

PERS-NR	NAME	VORNAME	,,,,,,,,,,,,,,,	.....	.....
.....	.....	.....	.....	.....	.....

Unformatierte Datenbestände

Hier können Sätze nicht durch Ordnungskriterien oder feste Stellenzuordnung der Felder identifiziert werden. Deshalb müssen wichtige beschreibende Schlagworte, sog. Deskriptoren[5] festgelegt (personell oder maschinell) werden.
Bei der Suche (Information Retrieval, IR) müssen die gespeicherten Deskriptoren mit den Suchbegriffen des Benutzers verglichen werden (über Wortähnlichkeits-prüfungen, Begriffskombinationen, Synonyme und logische Operationen sind die möglichen Fundstellen einzukreisen).

Zusammenfassung

Den grundsätzlichen Unterschied zwischen formatierten und unformatierten Daten soll das folgende Bsp. nochmals herausstellen:

Formatiert

Feldname	Pers-Nr	Name	Vorname	Beruf	Fam.-Stand	Abt.
Daten (Feldinhalte)	4711	Maier	Hans	Bäcker	verh.	10

Unformatiert

Der Mitarbeiter Hans Maier hat die Personalnummer 4711, er ist Bäcker, arbeitet in der Abteilung 10 und ist verheiratet.

Es gibt Systeme für formatierte und Systeme für unformatierte Daten. Sie sind unter folgenden Bezeichnungen bekannt:

Systeme für formatierte Daten	Systeme für unformatierte Daten
Numerische Informationssysteme	Nichtnum. Informationssysteme
Fact Retrieval	Document Retrieval
Data Base Systems	Refernce Systems
Data Management Systems	Information Retrieval Systems
Information Managament Systems	Information Storage and Retrieval
File Managent Systems	

Dokumente sind in der Regel nicht strukturiert. Sie bestehen aus einer Menge von Worten. Eine umfassende Textverarbeitung ist daher langsam und außerdem teuer, da „Textdatenbanken“ von beträchtlichem Umfang sind. So sind auch hier strukturierte Verarbeitungsformen nötig, d.h. die Darstellung des Textes muß auf Formate zurückgeführt werden. Ist das möglich, dann können die Zugriffsformen formatierter Datenbanken angewendet werden. Gewöhnlich dient dazu eine Liste von ausgesuchten Schlüsselworten je Dokument. Die Schlüsselwörter beschreiben den Inhalt des Dokuments und ermöglichen, das Dokument von anderen Dokumenten zu unterscheiden. So ergibt die Auswahl der Schlüsselwörter eine Indexierung zu den Dokumenten. Sie kann personell oder maschinell (, d.h. automatisch mit Hilfe des Rechners) erfolgen.

[5] vgl. 1.2.2.1

1.2.3.3 Daten- und Speicherstrukturen

Datenstrukturen

1. Verkettete Systeme

Verbindungen zwischen Datensätzen werden realisiert durch:

- Aufnahme von Adreßverweisen in den jeweiligen Datensatz
- spezielle sequentielle Anordnung von Datensätzen im Speicher

Satzadressen	Pers.Nr.-	Kostenstelle	Gehalt	Kettungsfelder	
				Kostenstelle	Gehalt
1	100	10	1230.-	3+	10
2	106	20	850.-	6+	7
3	110	10	1900.-	7	8
4	111	30	1600.-	5+	9
5	117	30	1400.-	8	4
6	120	20	740	9	2+
7	121	10	870	X	1
8	124	30	2400	10	X
9	130	20	1600	20	3
10	133	30	1340	X	5

Abb. 1.2-3: Adreßverkettung

2. Invertierte Systeme

Verbindungen werden über spezielle "inverted files" (Indexe, Indextafel, Wörterbücher, Adressbücher) aufgebaut. Invertierte Systeme sind weitgehend unabhängig von Datenstruktur und Zugriffspfad, da über die „inverted files“ erst im Bedarfsfall die Beziehungen aufgebaut werden.

Satzadressen	Pers.-Nr.	Kostenstelle	Gehalt
1	100	10	1230.-
2	106	20	850.-
3	110	10	1900.-
4	111	30	1600.-
5	117	30	1400.-
6	120	20	740.-
7	121	10	870.-
8	124	30	2400.-
9	130	20	1600.-
10	133	30	1340.-

Primärindextabelle Pers.-Nr.	
Feldwert	Satzadr.
100	1
106	2
110	3
111	4
117	5
120	6
121	7
124	8
130	9
134	10

Sekundärindextabelle Kostenstelle	
Feldwert	Satzadr.
10	1, 3, 7
20	2, 6, 9
30	4, 5, 8, 10

Sekundärindextabelle Gehalt	
Feldwert	Satzadr.
740.-	6
850.-	2
870.-	7
1230.-	1
1340.-	10
1400.-	5
1600.-	4, 9
1900.-	3
2400.-	8

Abb. 1.2-4: Indizierte Datei

Speicherstrukturen

Die Informationen müssen in einer Datenbank permanent gespeichert werden. Dateien spielen daher in Datenbanken eine große Rolle. Die Ausführungen über invertierte Systeme haben gezeigt, daß diese Daten nicht in einer einfachen Satzstruktur aufgebaut sind. Sogar Informationen über die Struktur der Daten müssen in Dateien gespeichert werden. Grundlage für Datenstrukturen sind Blöcke (kleinste physikalische Bearbeitungseinheit für fast alle Datenbank-Komponenten.). Die Größe eines Datenbankblocks ist nicht generell festgelegt. Sie kann beim Anlegen der Datenbank den Gegebenheiten (Anforderungen, Anwendung und Betriebssystem) angepaßt werden (gängige Werte: 2, 4 und 8Kbytes).

Blockadresse	Satzadresse	Pers.-Nr.	Kostenstelle	Gehalt
1	1	100	...	...
	2	106	...	...
	3	110	...	...
	4	111	...	...
2	5	117	...	...
	6	120	...	...
	7	121	...	...
	8	124	...	...
3	9	130	...	...
	10	133	...	...
	11	.....	...	...
	12	.....	...	...

Ist die DB einmal angelegt, dann ist die Blockgrösse eine nicht mehr änderbare Größe für die Datenbank. Für Dateien, die Daten der Datenbank enthalten, gilt demnach: Ihre Größe entspricht immer einem Vielfachen der Datenbank-Blockgröße.

Datenbank-Blöcke stellen die kleinste Verwaltungseinheit für den Speicher dar. Ein Datenbank-Block besteht immer aus einem Kopf und dem Inhalt (z.B. den Daten-sätzen einer Datei bzw. Tabelle).

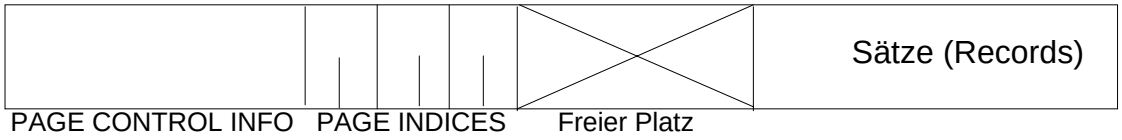


Abb. 1.2-5: Aufbau eines Datenbank-Blocks

Die logischen Strukturen der Datenbank (z.B. Tabellen und Indexe) werden in Dateien als (Datenbank-) Segment abgebildet. Jedes einzelne Segment besteht aus einer Anzahl von Blöcken (Seiten, Pages). In jedem Bereich sind die Blöcke fortlaufend numeriert, auch die Bereiche sind bekannt. Alle Datensätze der Datenbank können ermittelt werden, falls die Blocknummer des Blocks, in dem die Daten gespeichert sind, und das Segment, in dem sich der Block befindet, bekannt sind (physikalischer Databasekey).
Segment-Typen werden in drei Gruppen unterteilt:

1. Segment-Typen, die intern benötigt werden
2. Segment-Typen, die Benutzer-Daten (Dateien, Tabellen) aufnehmen
3. B*-Indexe, die zum schnellen Auffinden der Daten oder Sortierfunktionen genutzt werden.

Zur Speicherung und Verwaltung der Indexe werden B*-Bäume verwandt:

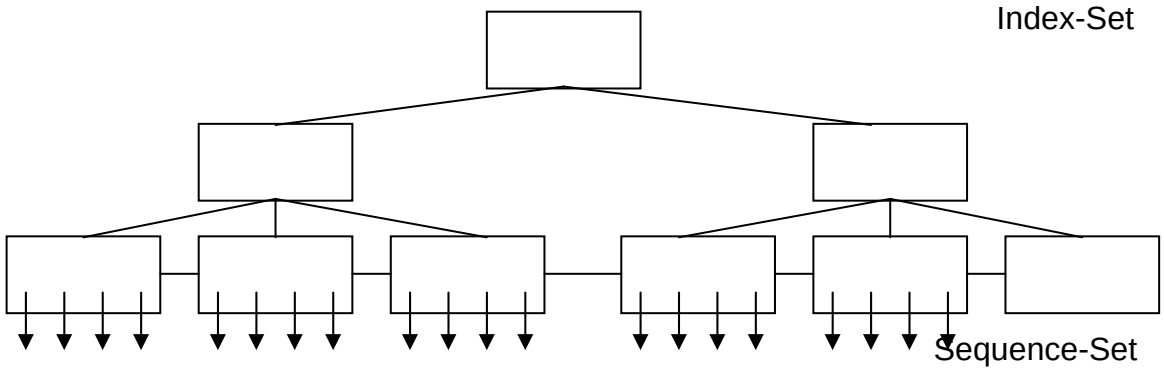


Abb. 1.2-6: Aufbau eines B*Baums

Die Knoten eines B*-Baums lassen sich in den sog. `Sequence Set` (Bereich der Bätter), der Schlüssel mit den Adressen der zugehörigen Datenblöcke in sortierter Reihenfolge enthält und den `Index Set` aufteilen, der den schnellen, gezielten Zugriff auf die Blöcke des `Sequence Set` ermöglichen soll. Ein typischer Zugriff auf einen B*-Baum beginnt bei der Wurzel und arbeitet sich über die Zeiger des Index Set an die betreffenden Knoten des `Sequence Set` heran. Dort werden die dem gesuchten Schlüsselwert entsprechenden Satzadressen gelesen (Databasekeys) und der Zugriff auf die Blöcke der zugeordneten Tabellen (Dateien) durchgeführt.

1.2.3.4 Grundfunktionen der Datenbanksoftware

Betriebssysteme bieten nur Zugriffsmethoden zu einfachen Dateien an. Zur Datenbank gehört demnach Verwaltungssoftware mit den Aufgaben:

- Verwaltung der Speicherstruktur
- Entfernen bzw. Einordnen der gewünschten Daten.

Man spricht von einem Datenbankverwaltungs-Programm oder **Data Base Management System** (**DBMS**).
Die Nutzung eines DBMS kann auf 2 Arten erfolgen:

1. Die Nutzung des DBMS findet zwischen Endbenutzer und dem DBMS auf direktem Wege statt. Es handelt sich dabei um ein geschlossenes System aus Datenbank und Daten-bankverwaltung. Die Systeme werden self-contained, exekutiv oder anwen-dungsorientiert genannt.
2. Die Kommunikation mit der Datenbank erfolgt über Programme. Die Bedienung eines solchen eingebetteten oder "host-language"-Systems erfordert Programmierkenntnisse, denn Datenbankanweisungen und Datenbank-beschreibung richten sich nach den Konventionen der Gastsprache der Benutzerprogramme. Man spricht von programmorientierten, operierenden bzw. "host-language" - Systemen.

Weiterhin muß

- es eine Methode zur Strukturdefinition der Datenbank (Design) geben.
- das Design in einer Struktur-Definitions-Sprache (**Data Description Language**, **DDL**) als Quellcode-Schema deklariert werden. Es muß ein DDL-Übersetzer (DDL-Prozessor) zur Verfügung stehen, um Objektcode zu erzeugen.
- es Ladebefehle (Laden der Datenbank) bzw. allg. "Utilities" zurAdministration geben
- eine Redefinition der Struktur leicht möglich sein
- zur Ein-/Ausgabe von Daten der Datenbank eine Möglichkeit zur Daten-manipulation vorhanden sein. Diese Möglichkeit kann in einer **Datenmanipulations-Sprache** (**DML**) vorliegen, die in eine gastgebende, herkömmliche Programmiersprache (Cobol, Pascal, C) eingebettet ist. Es kann aber auch eine eigenständige (meistens abfrageorientierte) Sprache (QL) vorliegen. In vielen Datenbanksystemen gibt es gleichzeitig beide Möglichkeiten.

Zu diesen Datenbankaufgaben kommt noch als weitere Aufgabe hinzu: System-steuerung zur Koordinierung der verschiedenen Aufträge innerhalb des Datenbanksystems und Datenkommunikation zum Nachrichtenaustausch mit Außenstellen (**Transaktionsbetrieb**).

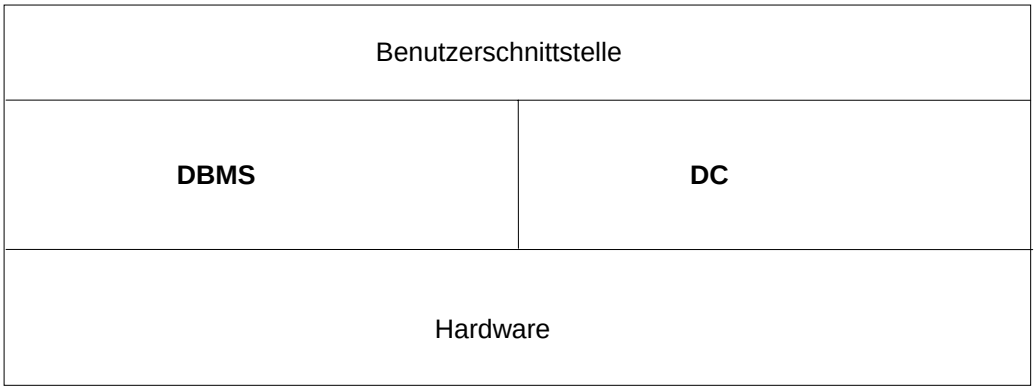


Abb. 1.2-7: Überblick zu den Komponenten eines DB/DC-Systems

Aufgaben eines DC-Systems

- Bereitstellen einer Schnittstelle für Anwender-/Datenbankprogramm, die die Eigenheiten der Datenstationen und die techn. Eigenschaften der Datenübertragung verdeckt.
- Abwicklung des Nachrichtentausches zwischen einer Vielzahl von Datenstationen und den Anwender-/Datenbankprogrammen
- Weiterleitung der Nachrichten in Abhängigkeit von Priorität, Betriebsmittel-verfügbarkeit und Schutzregelungen an den Empfänger

Der **Fernzugriff auf Datenbanken** ist im wesentlichen durch 4 Komponenten bestimmt:

- den Bildschirmarbeitsplatz
  Das kann bspw. ein Bankautomat mit einfacher Benutzeroberfläche oder ein komplexes System (CAD-Workstation mit leistungsfähigem Prozessor) sein
- das Anwendungsprogramm
  Das ist in der Regel ein transaktionsorientiertes Programm[6], das einfache Abfragen und Änderungswünsche entgegennimmt, Zugriffsberechtigung des Benutzers prüft, Datenbankzugriffe abwickelt und dem Benutzer das Resultat übermittelt.
- das Datenbanksystem (DB-System, DBS)
  Es besteht in der Regel aus mehreren Prozessen. Diese Prozesse erhalten ihre Aufträge von Anwendungsprogrammen (, die auch als eigene Prozesse ablaufen). Man unterscheidet Prozesse, die spezielle Aufgaben ohne Zuordnung zu einezelnen Anwendern ausführen (Hintergrundprozezesse) von Prozessen, die direkt (1:1) den angemeldeten Benutzern zugeordnet sind (Schattenprozesse, "Dedicated Server-Prozesse). All diese Prozesse sind Prozesse des Server (Backend). Die Versorgung des Backend mit Aufgaben übernimmt mindestens ein Frontend-Prozeß. Frontend-Prozesse werden auch als Client oder Anwendung bezeichnet.

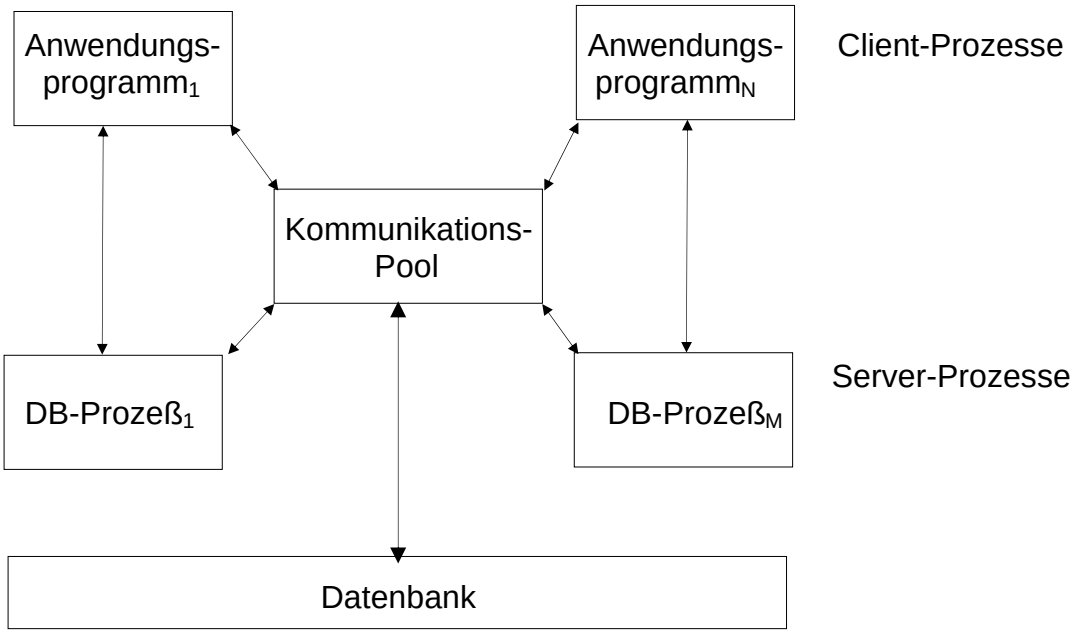


Abb. 1.2-8: Interprozeß-Kommunikation Anwender- bzw. Datenbank-Prozesse

Zur Abarbeitung gleichzeitiger Aktionen dient eine Datenverarbeitungszentrale im Hauptspeicher (Communication Pool, System Clobal area). Alle Daten, die das Datenbanksystem für mehrere verschiedene Aufgaben benötigen könnte, werden hier zwischengespeichert.
DB-Prozeße bearbeiten Transaktionen. Unter **Transaktion** versteht man eine Folge von Datenbankzugriffen, die eine Datenbank von einem gültigen Zustand in einen neuen, ebenfalls

[6] vgl. 1.5.2

gültigen Zustand überführen. Die Aktionen einer Transaktion sind aus Sicht des DBS die kleinsten ausführbaren (atomaren) Einheiten, deren korrekte Ausführung das DBS übernimmt.

- das Kommunikationssystem (DC-System)
  Es übernimmt die Verbindung der einzelnen Komponenten. Drei Fälle können unterschieden werden:

  1. Terminalnetze
     Beim Anschluß "dummer Terminals" über größere Entfernungen dient das Netz als "Verlängerungsschnur". Auf der Seite des Terminals müssen Kommunikations-protokolle bereitgestellt werden, die wegen ihres geringfügigen Aufwands in Hardware implementiert werden können. Auf der Seite des Rechners liegt der Schwerpunkt der Verarbeitung (z.B. Aufbau und Kontrolle der Verbindung, Program-mierung des Bildschirms).
  2. Netze zwischen autonomen Rechnern
     Zum Anschluß von "Intelligenten Bildschirmarbeitsplätzen" (z.B. PC, CAD-Arbeitsstation) mit integriertem Rechner muß ein Kommunikationsnetz zwischen eigenständigen Rechensystemen bereitgestellt werden.
  3. Verteilte Datenbanken
     Das Netz dient zur Kommunikation der einzelnen Datenbankverwaltungssysteme.

1.2.3.5 Data Dictionary / Directory System (DD/D-System)

Es enthält Informationen über Definition, Struktur und Benutzung von Daten. Das "`directory`" liefert dem eingesetzten DB-System Informationen über Plazierung und Strukturen innerhalb der Benutzerdatenbank (DB-System orientiert). Ein „**data dictionary**“ dagegen gibt dem Anwender Information über Daten, ihre Herkunft, wo sie genutzt werden und wann sie geändert wurden (zentralisierte Ablage von Informationen über Datendefinitionen).
Die meisten in einer Installation benötigten Daten befinden sich irgendwo in Bibliotheken, Dateien, Verzeichnissen oder Listen. In der Regel sind diese Informationen jedoch nicht hinreichend miteinander verbunden. Ein „**data dictionary** (**DD**)“ ist ein Hilfsmittel, bei dem alle Definitionen in (einer Gruppe von) Datenbanken gespeichert werden, um leichter Fragen beantworten zu können, Auswertungen durchführen und Ausgaben erzeugen zu können.

Ein DD enthält:

- Angaben über Definition, Herkunft, aktuelle Benutzung und Änderung, Struktur und Benutzungsvorschriften der Daten, z.B.: Namen, zulässige Wertebereiche, logische Beziehungen, Integritäts- und Zugriffsmöglichkeiten, Namen und Eigenschaften von Anwenderprogrammen mit Angaben über Speicherung, Codierung und Auffinden der Daten (Adreß und Längenangaben, Feldtypen, Zugriffspfade und physische Plazierung in der Datenbank)
- Angaben über Speicherung, Codierung und Längenangaben, Feldtypen, Zugriffspfade in der Datenbank.
- Angaben zu Verarbeitungseinheiten (Modul, Programm, Segment)

Funktionen eines DD sind:

- Erstellen von Berichten (z.B. Benutzerreports: welche Daten werden von wem benutzt?)
- Generierung von Datendefinitionen und Datenbank-Beschreibungen (Datenmodellierung) aus Definitionen im **DD**

Vorteile des **DD** bestehen auf vier Gebieten: Dokumentation, Datenbankdefinition, Anwendungsentwicklung und Kontrolle

Dokumentation
Wird das DD richtig eingesetzt, so wird es bspw. die Definition des Felds Pers.-Nr. nur einmal geben und diese Definition befindet sich im **DD**. Im **DD** werden Definitionen nicht mehrfach gespeichert, sondern entsprechend untereinander verknüpft.

Datenbankdefinition
Definition zu Datenbanken werden zweckmäßig im **DD** angelegt und verwaltet.

Anwendungsentwicklung
Das **DD** ist eine Hilfe festzustellen, ob bestimmte Datendefinitionen bereits im **DD** vorhanden sind oder nicht. Außerdem besteht die Möglichkeit Satzstrukturen im neutralen Format des **DD** anzulegen.
Kontrolle
Das **DD** ist eine Hilfe, Definitionen eindeutig zu machen.

Dienstprogramme zur Auswertung eines **DD** ermöglichen

- die Analyse der vorhandenen Datenobjekte (z.B.: die Bestimmung von Redun-danzen, Inkonsistenzen)
- "Cross-Reference- " Listen
- Statistiken
  (Benutzungshäufigkeiten, Werteverteilungen) zur Optimierung der Speicher-struktur

Ein **DD** läßt sich, falls alle Daten über Dateien und Programmsysteme in das Wörterbuch aufgenommen werden, zur zentralen Kontrollinstanz über alle gespeicherten Daten ausbauen.
Man spricht häufig auch von aktiven und passiven „Data Dictionaries“ und bezieht sich dabei auf den Grad der Integration des Data Dictionary mit dem Datenbankverwaltungssystem (DBMS). Falls das DBMS die Definition des **DD** zur Laufzeit von Programmen auf den neuesten Stand bringt, liegt ein aktives **DD** vor. Ist dazu ein eigenständiger Regenerationslauf nötig, handelt es sich um ein passives **DD**.
In beiden Fällen ist dann die Architektur einer Datenbank[7] um das Datenwörterbuch (**DD**) organisiert:

[7] vgl. 1.2.4.6

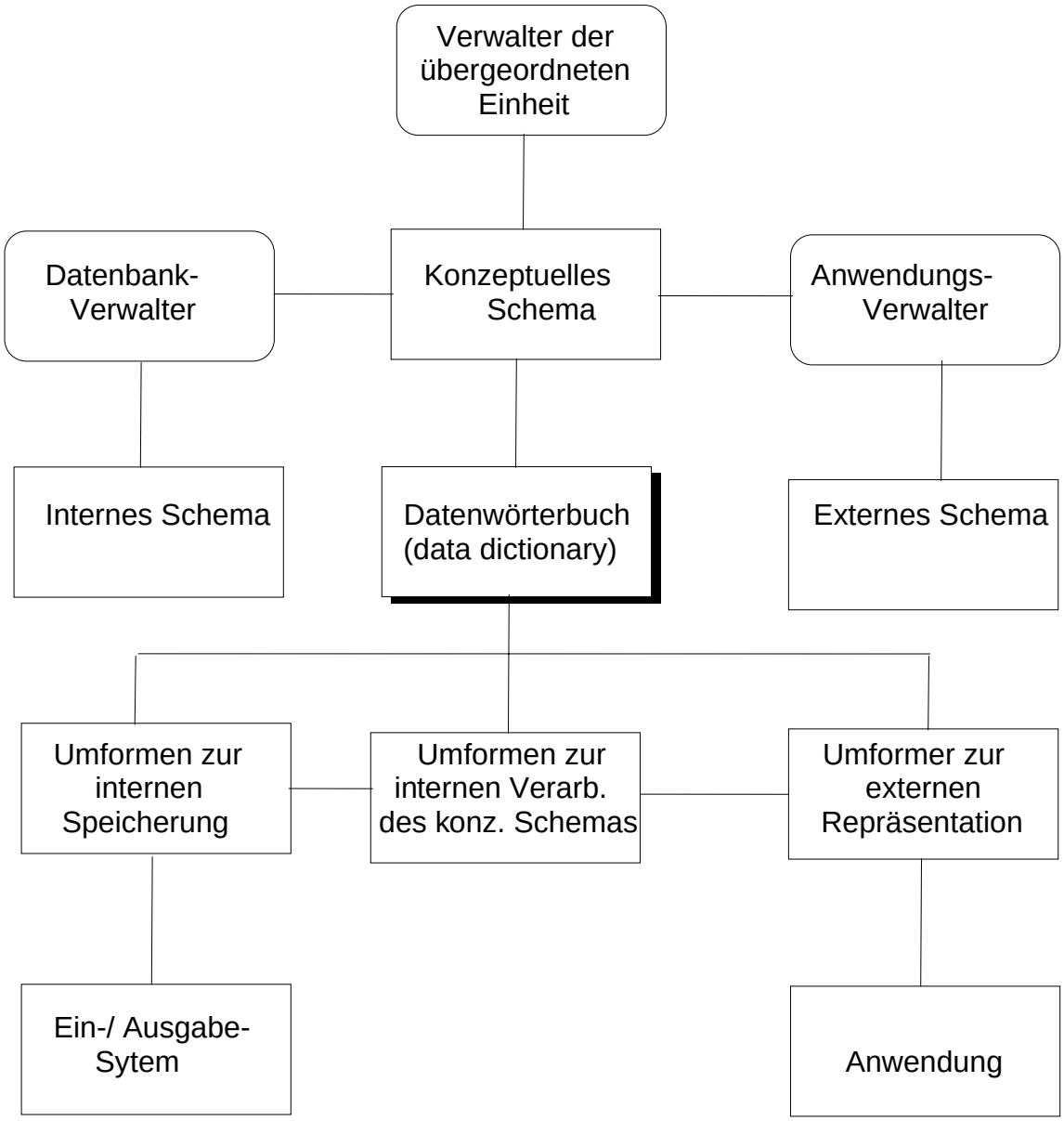


Abb.: 1.2-9: Organisation einer DB-Architektur um das DD

1.3 Datenbankmodelle für formatierte Datenbanken (DB)

Sie werden danach eingeteilt, wie die elementaren logischen Einheiten der Datenbank, die Datensätze, zu Datenstrukturen (Baum, einfaches Netz, Tabelle) zusammengefaßt sind.

1.3.1 Beschreibung der Daten in formatierten Datenbanken

1.3.1.1 Entitätsmengen und ihre Beziehungen

1. **Grundlagen**

Durch eine Datenbankanwendung sind die unterschiedlichen Aufgaben eines Bereichs zu koordinieren. Ein derartiger Bereich ist eine umfassende Verwaltungseinheit, z.B.:

- ein Industrieunternehmen (mit Produktionsdaten)
- eine Bank (mit Daten diverser Konten)
- ein Krankenhaus (mit Daten über Patienten)
- eine Hochschule (mit Daten über Studenten/ Dozenten)

Bsp: „Produktionsdaten eines Unternehmens“
Informationen werden gewünscht über

- die gegenwärtig vorhandenen Projekte
- die Bauteile/Teile, die zur Durchführung dieser Projekte benötigt werden
- die Lieferanten, die die Bauteile bereitstellen
- die Angestellten, die an diesen Projekten mitarbeiten

Projekte, Lieferanten, Bauteile, Angestellte sind die elementaren Einheiten, über die Daten in der Datenbank gespeichert sind.
Das ergibt folgenden Schemaentwurf:

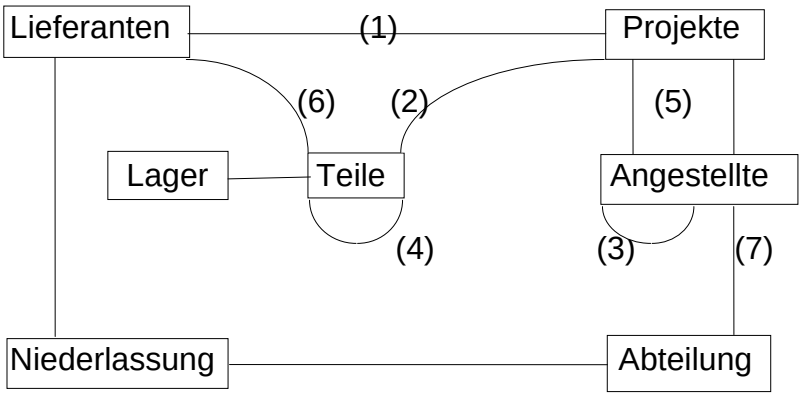


Abb. 1.3-1: Schema für die Produktionsdaten eines Unternehmens

Die Beschreibung eines Bereichs der realen Welt ist nur durch Abstraktion vieler konkreter Gegebenheiten und geeignete Modellbildung möglich.

Modellbildung in einem Datenbanksystem bedeutet:

Gewisse Dinge (**Objekte**) der realen Welt werden herausgestellt, da sie zur Problemlösung benötigt werden. Zwischen den Objekten existieren **Beziehungen**.

Anstatt des Begiffs Objekt benutzt man hier auch häufig den Begriff **Entität** (entity).[8]

2. **Was ist eine Entität**?

Eine **Entität** ist ein individuelles oder identifizierbares Exemplar von Dingen, Personen oder Begriffen der realen Welt.
Eine **Entität** kann demnach sein:

- ein Individuum (z.B. ein Student, ein Dozent, ein Mitarbeiter, ein Einwohner)
- ein reales Objekt (z.B. eine Maschine, ein Gebäude, ein Produkt)
- ein abstraktes Konzept (z.B. eine Fähigkeit, eine Vorlesung)
- eine Beziehung

Wird ein Individuum, ein reales Objekt, ein abstraktes Konzept, ein Ereignis oder eine Beziehung als Entität deklariert, so wird damit ausgedrückt: „Sachverhalte, die den als Entität deklarierten Begriff betreffen, sollen speicherbar sein, sobald ein eindeutig **identifizierendes Merkmal** bekannt ist."
Entitäten können in verschiedenen **Entitätstypen** klassifiziert werden, z.B. Studenten, Vorlesungen, Angestellte. Eine Entität wird durch ihre **Attribute** beschrieben. Einer Entität werden also Merkmale (Eigenschaften) zugeordnet, die zu seiner Beschreibung wichtig sind.

Bsp.: „Die Produktionsdaten eines Unternehmens"
Zur Beschreibung des Entitätstyp Angestellte ist notwendig: NAME, BERUFSBEZEICHNUNG
Zur Beschreibung des Entitätstyp TEILE kann herangezogen werden: BEZEICHNUNG, FARBE, KOSTEN, GEWICHT.
BEZEICHNUNG, FARBE, GEWICHT sind Eigenschaftsnamen (Attribut-Namen).

Jedes **Attribut** kann Werte aus einem bestimmten Wertebereich (**Domäne**) annehmen. Eine **Domäne** stellt die Menge aller möglichen Werte für eine spezifische Eigenschaft bereit, z.B.

```
NAME       ALTER GEWICHT BEURTEILUNGSKRITERIUM   STUDIENFACH
  Karl        20  66              gut              Mathematik
  Fritz       25  75              mittel           Chemie
  Juergen     27  68              schlecht         Informatik
  Josef       24  87                               Physik
```

Darstellung von Entitäten

Jedem Entitäts-Typ kann man eine Kombination von Attributen, jeder Entität dieses Typs eine entsprechende Kombination von Attributwerten zuordnen.
Bsp.: „Produktionsdaten eines Unternehmens"
Dem Entitätstyp ANGESTELLTE kann zugeordnet werden:

```
NAME: Karl
BERUFSBEZEICHNUNG: Programmierer
NAME_DER_KINDER: Fritz, Anna
```

In der Regel sind Attributkombinationen so beschaffen, daß sie die zugehörige Entität eindeutig beschreiben.
Alle Entitäten des gleichen Typs bilden eine Entitätsmenge

[8] vgl. Vetter, M.: Aufbau betrieblicher Informationssysteme, 6. Auflage Stuttgart 1990

Bsp.: "Alle Studenten einer Hochschule" (charakterisiert durch die Eigenschaften NAME, ALTER, WOHNORT)

Ein **Entitäts-Schlüssel** (Schlüsselkandidat, „`candidate key`“) ist dann ein Attribut, das Entitäten innerhalb einer Entitätsmenge eindeutig identifiziert. Ein Entitätstyp kann mehrere Schlüsselkandidaten haben. Es ist sinnvoll, aus den Schlüsselkandidaten einen **Primärschlüssel** („`primary key`“) auszuwählen. Soweit es möglich ist, sollte man zusammengesetzte Primärschlüssel vermeiden.

3. **Beziehungen**

Zwischen Entitäten können Beziehungen bestehen. So sind bspw. in den „Produktionsdaten eines Unternehmens“ Beziehungen:

(5) Ein Angestellter ist Mitarbeiter an bzw. ist Projektleiter von bestimmten Projekten
(7) Ein Angestellter gehört zu einer bestimmten Abteilung

Gleichartige Beziehungen können zu einer Beziehungsmenge (bzw. zu einem Beziehungstyp) zusammengefaßt werden.

4. **Klassifizierung von Beziehungen**

Es können identifizierende oder beschreibende Attribute unterschieden werden.

Ist E bzw. E' die Menge der identifizierenden Attributwerte und W die Menge der be-schreibenden Attributwerte, dann existieren folgende Abbildungsmöglichkeiten:

1) Beziehungen zwischen identifizierenden und beschreibenden Attributwerten ($E \rightarrow W$).
2) Beziehungen zwischen identifizierenden Attributwerten verschiedener Entitätsmengen ($E \rightarrow E'$)
   Bsp.: „Produktionsdaten eines Unternehmens“
   (6) Lieferanten-Nr. $\rightarrow$ Teile-Nr.
   Ein Vorkommen des Lieferanten L1 identifiziert bspw. eine bestimmte Anzahl (0 inbegriffen) von Teilen T1, T2, T3.
   Ein Vorkommen T2 identifiziert bspw. eine bestimmte Anzahl von Lieferanten (L1, L3).
3) Beziehungen zwischen identifizierenden Attributwerten der gleichen Menge ($E \rightarrow E$).
   Bsp.: "Produktionsdaten eines Unternehmens"
   (3) Unterstellungsverhältnis
   Es zeigt, welche Angestellte Vorgesetzte bzw. Untergebene sind.
   (4) Stückliste
   Sie zeigt, welche Unterteile (Komponenten) montiert werden, um ein Oberteil zu erhalten
   (4) Verwendungsnachweis
   Er zeigt, in welche Oberteile (Baugruppen) eine Komponente eingeht.

Häufig ist die Einteilung komplexer Strukturen ($E \rightarrow E$, $E \rightarrow E'$) abhängig von der Abgrenzung der Entitätsmengen., z.B.:

a) Die Beziehung Pers.-Nr. des Ehemanns zur Pers.-Nr. der Ehefrau ist von der Art $E \rightarrow E$, wenn alle Personen einer Menge zusammengefasst werden.
b) Erfolgt die Zusammenfassung getrennt nach Männern und Frauen, so liegt die Art $E \rightarrow E'$ vor.

5. **Kardinalitätsverhältnis**

Es gibt darüber Auskunft, in welchem Verhältnis zwei Entitätstypen über einen bestimmten Beziehungstyp miteinander verbunden sind.
Man unterscheidet:

Beziehungen vom Verhältnis 1:1
Zwischen zwei Entitätstypen (Entitätstyp 1 und Entitätstyp 2) kann zu einem Zeitpunkt nur eine einzige Entität aus Entitätstyp 1 mit einer einzigen Entität aus Entitätstyp 2 in Verbindung stehen. So ist der Beziehungstyp "Heirat" eine 1:1 - Verbindung zwischen zwei "Personen"-Entitäten.
Beziehungen vom Verhältnis 1:N
Zwischen den beiden Entitätstypen kann zu einem Zeitpunkt nur eine einzige Entität aus Entitätstyp 1 mit mehreren Entitäten aus Entitätstyp 2 in Verbindung stehen. So ist bspw. der Beziehungstyp zwischen Abteilung und Angestellte[9] vom Verhältnis 1:N.
Beziehungen vom Verhältnis M : N
Zwischen zwei Entitätstypen können zu einem Zeitpunkt mehrere Entitäten aus Entitätstyp 1 mit mehreren Entitätstypen aus Entitätstyp 2 (über den Beziehungstyp) in Verbindung stehen.
So ist bspw. der Beziehungstyp zwischen Projekte/Teile vom Verhältnis M:N.

6. **Mitgliedsklassen**

Ergibt sich aus den Anforderungen des Anwendungsgebiets, daß jedes Exemplar eines Entitätstyps an einer Beziehung teilhaben muß, dann wird diese Klasse des Entitätstyps in dieser Beziehung als "**mandatory** (zwingend)" bezeichnet. Andernfalls ist die Mitgliedsklasse "**optional** (freigestellt)".
Bsp.: Eine Datenbank für die Verwaltung einer Gemeinde umfaßt Bürger (alle Einwohner) und Haushaltsvorstände (zweckmäßigerweise sind das die Einwohner der Gemeinde von denen man Steuern erheben kann). Jeder Bürger ist zwingend mit dem Datum der Geburt Einwohner der Gemeinde und bleibt das, bis er "gelöscht" wird (z.B. beim Tod oder einem Umzug). Zum Steuerzahler wird er dann (im übertragenene Sinne optional), falls er Grundbesitz (Grundsteuer) hat oder ein Gewerbe betreibt.
Die Entscheidung, ob eine Mitgliedsklasse eines Entitätstyps in einer Beziehung zwingend oder freigestellt ist, liegt häufig im Ermessen des Datenbank-Modellierers.

1.3.1.2 Beziehungen und Beziehungsattribute

Ein Beziehungsattribut beschreibt die Verknüpfung einer Beziehungsmenge mit einer Domäne bzw. mehreren Domänen, z.B.:

Beziehungen zwischen Studenten (repräsentiert durch den Entitätsschlüssel `S#` mit den Erscheinungsformen `{S1, S2, S3}` und Dozenten (repräsentiert durch den Entitätsschlüssel `D#` mit den Ausprägungen `{D1, D2}` werden durch die

[9] vgl. Abb. 1.3-1

Beziehungs-menge BELEHRT beschrieben, der das Beziehungsattribut BEURTEILUNG zugeordnet werden kann. So kann bspw.

```
BELEHRT
   (D1, S1)
   (D1, S3)
   (D2, S1)
   (D2, S2)
   (D2, S3)
```

über das Beziehungsattribut Werte aus dem Wertebereich KRITERIUM gut, mittel, schlecht zugeordnet erhalten.
Das Attribut BEURTEILUNG verknüpft die Beziehungen BELEHRT mit der Domäne (Beurteilungs-) KRITERIUM. Ein spezifisches Dozenten-Studenten-Beziehungspaar steht mit einem einzigen spez. Kriterium in Beziehung (Assoziation einfacher Art).Beziehungen werden identifiziert, indem man die Schlüssel der Entitäten in der Beziehung nutzt.

1.3.2 Das relationale Datenbankmodell

1.3.2.1 Grundlagen

Grundlage dieses Datenmodells sind einfache Tabellen, z.B.:

Schlüssel	Name	Vorname	Fakultät	Geburtsdatum
110506	.....	Werner	Medizin	10.03.71
060313	.....	Karl	Jura	10.01.73
2517008	.....	Fritz	Mathematik	09.03.70

Abb. 1.3-2: Schematische Darstellung zu einem relationalen DB-Modell

Die Tabelle wird auch **Relation** genannt. Datenbanken, die aus solchen Relationen aufgebaut sind, heißen **relationale Datenbanken**. Die Datenbanken bestehen aus "flachen", hierarchiefreien Zusammenstellungen von Datenfeldern. Die Spalten der Tabelle bezeichnet man als Attribute der Relation. Eine flache Datei (flat file) ist eine Relation und besteht aus einer Menge von Tupeln (Tabellen-Zeilen). Zu einem Tupel sind die auf ein bestimmtes Objekt bezogenen Datenwerte zusammengefaßt.
Das relationale Datenbankmodell[10] besitzt im wesentlichen folgende Eigenschaften[11]:

1) Die Daten werden einheitlich durch Werte repräsentiert, die in Form von Tabellen dargestellt werden.
2) Der Benutzer sieht keine speziellen Verweisstrukturen zwischen den Tabellen.
3) Es gibt Operationen zur Auswahl von Tabellenzeilen (Selektion), Tabellenspalten (Projektion) sowie zur Verbindung ("join") von Tabelleneinträgen. Keine dieser Operatoren ist jedoch auf Kontrollstrukturen angewiesen oder durch vordefinierte Zugriffsstrukturen beschränkt. Diese

[10] vgl. Codd, E.F.: "A Relational Modell of Data for Large Shared Data Banks", Communications of the ACM, June 1970, Seiten 377 - 387

[11] Datenbanksysteme, die auf diese Konzepte beschränkt sind, werden als "minimal relational" bezeichnet.

Operationen werden über eine Datenmanipulationssprache (**DML**[12]) realisiert, die auf Daten zugreifen, Daten einfügen, löschen, korrigieren und Anfragen beantworten kann. Datenmanipulationssprachen im Bereich relationaler Datenbanken sind besonders leicht und einfach. Ein weit verbreitetes Beispiel dafür ist SQL[13].

Die Darstellung der Daten in einer relationalen Datenbank folgt speziellen Vorschriften. Diese Vorschriften (Theorie des relationalen Datenbankentwurfs) bezeichnet man üblicherweise als **Normalisierungstheorie**[14]. Normalisieren bedeutet: Darstellung des logischen Schemas einer relationalen Datenbank in der Form einfacher, nicht geschachtelter Tabellen.
Das relationale Datenbankmodell ist heute die übliche Organisationsform, in der Daten für die Abbildung im Rechner beschrieben werden. Für die Darstellung gelten allgemein folgende Regeln:

- Alle Datensätze (Tabellenzeilen) sind gleich lang
- Jeder Datensatz (Tupel) kommt nur ein einziges Mal in einer Tabelle vor, die Reihenfolge der Sätze (Tupel) ist beliebig. Jede Tabellenzeile beschreibt einen Datensatz. Die eindeutige Identifizierung eines Tupels erfolgt über den „Schlüssel“
- Der Wert eines Attributs kommt aus einem bestimmten Wertebereich (Domäne). Jede Tabellenspalte beschreibt eine bestimmte Eigenschaft (Attribut) des Datenobjekts.
- Durch Kombination einzelner Spalten (Attribute) und Zeilen (Datensätze) können neue Tabellen (Relationen) gebildet werden.

1.3.2.2 Normalisieren

Im wesentlichen heißt **Normalisieren** Beseitigung der funktionalen Abhängigkeiten, die zwischen den einzelnen Attributen eines Relationenschemas vorliegen können. Eine **funktionale Abhängigkeit** ist zwischen Attributmengen, z.B. A_i und A_j dann gegeben, falls in jedem Tupel der Relation der Attributwert unter A_i -Komponeneten den Attributwert unter den A_j-Komponenten festlegt. Die funktionale Abhängigkeit wird dann so beschrieben: $A_i \rightarrow A_j$.

Schlüssel sind Spezialfälle funktionaler Abhängigkeiten. Ein Schlüssel X beschreibt für ein Relationenschema eine funktionale Abhängigkeit, wenn X minimal ist (d.h. aus der kleinsten Menge Tupel identifizierender Attributwerte gebildet ist). Ziel eines sich auf Abhängigkeiten abstützenden Datenbankentwurfs einer relationalen Datenbank ist: Umformen aller funktionalen Abhängigkeiten in Schlüssel-abhängigkeiten. Die Menge der Abhängigkeiten ist äquivalent zur Menge der Schlüsselbedingungen im resultierenden Datenbankschema (**Abhängigkeitstreue**).
Bsp.: Gegeben sind in der folgenden Tabelle die Studiendaten von Studenten an einer Hochschule (, Schlüssel sind MATNR und SRNR).

STUDIENDATEN[15]

MATNR	NAME	ADRESSE	SRNR	STUDR.	ANFANGSDAT
123	Meier	Berggasse 19	88	Informatik	01.10.1989
124	Müller	Lange Gasse 19	81	Physik	01.03.1989

[12] vgl. 1.2.3.4
[13] vgl. 1.4.3.2
[14] vgl. 2.1.1
[15] Abkürzungen: MATNR...Matrikelnummer SRNR...Studienrichtungsnummer
STUDR...Studienrichtung ANFANGSDAT...Anfangsdatum

123	Meier	Berggasse 19	79	Psychologie	01.10.1985
125	Schmidt	Grabenweg 4	79	Psychologie	01.03.1990
128	Lang	Brückenweg 23	88	Informatik	01.10.1989
129	Lang	Brückenweg 23	81	Physik	01.10.1989
127	Bauer	Hochstraße	88	Informatik	01.10.1977

Abb. 1.3-3: Tabelle mit Studiendaten

Die vorliegende Tabelle zeigt redundante Daten (Informatik, Physik, Psychologie). Die Abhängigkeiten dieser Daten (Bezeichnung der Studienrichtung) vom Attribut Studienrichtungsnummer ist offensichtlich, zweckmäßigerweise ist diese Abhängigkeit in einer eigenen Tabelle zu verwalten. Die Attribute NAME, ADRESSE sind vom Teilschlüssel MATNR abhängig, die Attribute STUDR, ANFANGSDATUM vom Teilschlüssel Studienrichtungsnummer abhängig. Es ist besser, solche Abhängigkeiten in getrennten Tabellen zu verwalten.
So führt die Normalisierung der vorstehende Tabelle auf folgende Tabellen:

STUDENT

MATNR	NAME	ADRESSE
123	Meier	Berggasse 19
124	Müller	Lange Gasse 19
125	Schmidt	Grabenweg 4
127	Bauer	Hochstraße 2
128	Lang	Brückenweg 23
129	Lang	Brückenweg 23

STUDIUM

MATNR	SRNR	ANFANGSDAT
123	88	01.10.1989
124	81	01.03.1989
123	79	01.10.1985
125	79	01.03.1989
127	88	01.10.1977
128	81	01.10.1989
129	88	01.10.1989

STUDIENRICHTUNG

SRNR	STUDR
88	Informatik
81	Physik
79	Psychologie

Abb. 1.3-5: Normalform der Tabelle mit Studiendaten

Der vorliegende Zerlegungsprozeß zeigt eine geeignete Auswahl von Spalten der Ausgangstabelle. Normalisieren heißt demnach auch: „Zerlegen von Relationenschemata in kleinere, übersichtlichere Tabellen“. Es sind nur sinnvolle Zerlegungen (Kriterium: funktionale Abhängigkeiten) zugelassen, d.h.: Die Ausgangsrelation muß sich (ohne Informationsverlust) durch Zusammenfügen von Teilrelationen der Zerlegung rekonstruieren lassen (**Verbundtreue**). Zerlegungen sind nur dann sinnvoll, wenn sie verlustfrei sind. Beim Zusammensetzen der Teilrelationen müssen wieder genau die Tupel der Ausgangsrelation erzeugt werden. Es sollen aber möglichst wenige Tabellen erzeugt werden, die den Forderungen genügen (**Minimalität**).

Eine Menge funktionaler Abhängigkeiten beschreibt die Integritätsbedingungen über dem Relationenschema. Sie wird bei der Zerlegung des Relationenschemas mit folgender Einschränkung auf die Schemata der Tabellen vererbt: Über einem Teilschema dürfen nur Abhängigkeiten definiert sein, deren Attribute vollständig im Teilschema enthalten sind. Zum anderen sollte die Menge der Abhängigkeiten, die über dem Schema gültig sind, zusammengenommen äquivalent zu der Menge über dem Ausgangsschema sein (**abhängigkeitsbewahrende Zerlegung**).
Ein anderer Vorteil der Zerlegung größerer Tabellen in kleinere Tabellen ist die leichtere Definition von Sichten (Views). Ein „**View**“ ist eine Relation, die nicht explizit in der Datenbank gespeichert ist, sondern aus den gespeicherte Tabellen abgeleitet wird. Nachdem ein „View“ definiert wurde, kann er in gleicher Weise benutzt werden wie jede andere Relation.
Die Suche in einer relationalen Datenbank erfolgt durch Angabe von bestimmten Attributwerten (eingebettet in Abfragesprachen). Da das Durchsuchen aller Datensätze viel zu lang dauern würde, muß die für das Suchen wichtige Information über einen Index herausgezogen werden. Im Index steht neben der Suchinformation dann ein Verweis auf den vollständigen Datensatz. Selbstverständlich ist der Index so einzurichten, daß die Suchzeit möglichst kurz ist. Außerdem belegt der Index zusätzlichen Speicherplatz. Man kann natürlich auch mehrere Indexe für verschiedene Suchkriterien einrichten. Die Architektur einer relationalen Datenbank ist somit intern wesentlich durch die Organisation der Daten- und Indexdatei(en) bestimmt.
Durch Integration der logischen Programmierung (Prolog) mit relationalen Datenbanken entstehen **deduktive Datenbanken**. Sie ergänzen die Fakten der relationalen Datenbank (Tupel) mit Regeln. Relationale Datenbanken werden dadurch zu Wissensbasen. Die zur Manipulation logischer Datenbanken entwickelten Datenbanksprachen (z.B. Datalog) stützen sich auf die **Prädikatenlogik**.

1.3.3 Das Entity-Relationship Modell

Grundlagen

Das Entity-Relationship Modell [16] von Chen ist ein Versuch, Beziehungen zwischen den Daten zu formalisieren. Die Objekte der Wirklichkeit, die durch die Daten modelliert werden, heißen "Entitäten (`entities`)". Dies können beliebige Objekte sein, z.B.: reale Gegenstände, gedankliche Einheiten, Vorgänge [17].
Es gibt zwei Arten von Relationen:

1. Entitäts-Relationen (**Entitätsmengen**)
Sie beschreiben die Eigenschaft einer Klasse von Entitäten, d.h. eines Entitätstyps.
2. "Relationship" - Relationen (**Beziehungsmengen**)
Sie verknüpfen 2 (oder mehr) Entitäten und beschreiben diese Verknüpfung mit Attributen, die nur dieser Beziehung zugeordnet werden können.

Es gibt zahreiche Varianten des Entity-Relationship Modells[18].All diese Modelle dienen in erster Linie theoretischen Überlegungen oder der Strukturierung von Datensammlungen (konzeptioneller Entwurf), um Verknüpfungen deutlich zu machen und danach Dateien in einem der üblichen Datenbanksysteme sinnvoll definieren und einrichten zu können.
Mit Entitäts- und Beziehungsmengen läßt sich jeder Ausschnitt der realen Welt (Miniwelt) beliebig detailliert beschreiben. Eine Beziehung kann dabei auch aus mehr als zwei Entitätsmengen bestehen.

Bsp.: Die Beziehung "hoert" zwischen "Student", "Vorlesung" und "Professor" ist eine dreistellige Beziehung.

Das Entity-Relationship Modell stellt den Rahmen für die Angabe derartiger mehrstelliger Beziehungen bereit (konzeptioneller Entwurf). Natürlich kennt das Entity-Relationship Modell auch die in der realen Welt häufig vorkommenden binären Relationen, z.B.:

1. Die Beziehung zwischen Ehemann und Ehefrau ist vom Typ 1 : 1.
2. "Professoren können mehr als eine Diplomarbeit (gleichzeitig) betreuen, eine Diplomarbeit muß von genau einem Professor betreut werden". Das ist eine hierarchische "1 : n"-Beziehung.

Gelegentlich stehen Entitätsmengen auch über eine "ist_ein (IS_A)"-Beziehung in Verbindung. So ist bspw. ein Professor an einer Hochschule ein "Hochschulangestellter". "Hochschulangestellte" sind aber auch die Verwaltungsangestellten. Der Professor an einer Hochschule besitzt neben der Identifikation, die ihn als Hochschulmitarbeiter ausweist und einer Reihe weiterer Attribute (Name, Geburtsdatum, Adresse) Merkmale, die ihn als Mitglied eines Fachbereichs in einer bestimmten Hochschule (z.B. Fachhochschule Regensburg) ausweisen.

---

[16] vgl. Chen, P., S.: The Entity-Relationship-Modell: toward a unified view of data, ACM Transactions on Database Systems, March 1976

[17] vgl. 1.3.1.1

[18] Es spielt auch eine wesentliche Rolle bei den Werkzeugen des computerunterstützten Software-Engineering (CASE)

Die zweistellige Beziehung "E_1 ist_ein E_2" besagt: E_1 ist eine Spezialisierung von E_2 (bzw. E_2 ist eine Verallgemeinerung von E_1). Im vorliegenden Fall könnte E_1 (Professoren) auch als Teilmenge von E_2 (Hochschulmitarbeiter) verstanden werden. Die Darstellung von E_1 und E_2 als verschiedene Mengen ermöglicht es jedoch, Besonderheiten von E_1 zu modellieren, die nicht unbedingt für jede Entität aus E_2 relevant sind.

Darstellung

Das Entity-Relationship Modell dient vor allem zur Beschreibung des konzeptuellen Schemas[19] einer Anwendung. Die Struktur aus Entitätsmengen, Beziehungsmengen und Attributen wird im **E**ntity-**R**elationship-Diagramm (**ER**-Diagramm) grafisch [20] dargestellt:

1. Deklaration von Entitätsmengen
Sie erfolgt über ein Rechteck, das den Namen der Entitätsmenge enthält, und durch Kreise, die die Attribute aufnehmen. Die Kreise werden durch ungerichtete Kanten mit dem Rechteck verbunden. Elemente des Primärschlüssel werden unterstrichen, Wertebereiche werden nicht dargestellt.
2. Deklaration von Beziehungsmengen
Sie erfolgt durch eine Raute [21], die den Namen der Beziehungsmenge enthält. Die Raute wird durch Kanten mit der beteiligten Entitätsmengen-Deklaration (Recht-ecke) verbunden. Die Kanten sind nicht gerichtet (Ausnahme: Hierarchische Beziehungen). Der Typ der Beziehung wird an die Kante geschrieben. Falls die Beziehungsmenge Attribute besitzt, werden diese ebenfalls durch Kreise dargestellt und über (ungerichtete) Kanten mit der entsprechenden Raute verbunden.

Bsp.: Beziehung zwischen Lieferant und Artikel im ERM

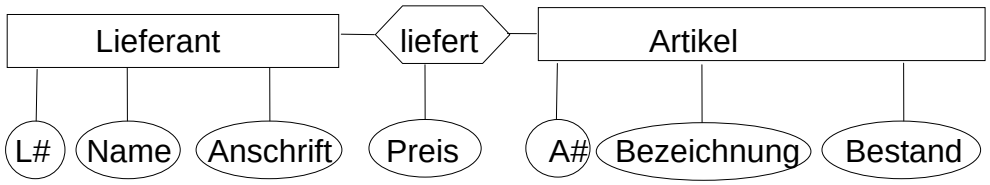


Abb. 1.3-5: ERM-Diagramm zur Beziehung "Lieferant-Artikel"

Zur Darstellung der Ausprägungen (Mengen, Occurrences) greift das ERM häufig auf das relationale Datenbankmodell zurück. Für jeden Entitätstyp und jeden Beziehungstyp wird eine eigene Tabelle angelegt. Jedes notwendige Attribut erhält in einer solchen Tabelle eine Spalte zugeordnet. Bei den Tabellen der Beziehungstypen muß darauf geachtet werden, daß auch die Primärschlüssel der über die Beziehungstypen verknüpften Entitätstypen vorliegen.

[19] vgl. 1.3.6

[20] vgl. Chen, P. S. und Knöll, Heinz-Dieter: Der Entity-Relationship-Ansatz zum logischen Systementwurf, Mannheim/Wien/Zürich, 1991

[21] Sie wird zur besseren Übersicht generell abgeflacht dargestellt

Lieferant		
L#	Name	Anschrift
L01	....	....

liefert	
L#,A#	Preis
L01,A17	.....

Artikel		
A#	Bez	Best
A17	....	....

Abb. 1.3-6: Relationale Datenbank zur Beziehung "Lieferant-Artikel"

Die Komplexität eines Beziehungstyps gibt an, in welchem Verhältnis die Entitäten der beteiligten Entitäts-Typen zueinander in Beziehung stehen. Sie wird durch die Beschriftung der Kanten (1, m, n) ausgedrückt. Zwischen 2 Entitäts-Typen können Beziehungen des Typs 1:1, 1:n (eins zu viele) und m:n (viele zu viele) vorkommen[22]. Einer Kante kann ein Rollenname zugeordnet sein, der die Funktion des jeweiligen Entitäts-Typen in Bezug auf den Beziehungstyp beschreibt.

„**ist_ein**"-Beziehung: Sie wird durch eine Raute mit der Beschriftung „ist_ein" oder durch einen Pfeil dargestellt. Die Kante zu E_1 einer Beziehung "E_1 ist_ein E_2" ist ungerichtet, die Kante zu E_2 gerichtet.

Bsp.: Entity-Relationship Diagramme zu einer Hochschulverwaltung

Entitätsmengen[23] sind

```
Hochschulmitarbeiter
(MNR, Gebdat, Adr, ...)
MNR ... Mitarbeiternummer [24]

Diplomarbeit
(Thema, Beginn, Ende, Note)
Student
(Matrikelnummer, Name, Vorname, .... )
Vorlesung
(Name, ....)
Professor
(MNR, Fachbereich, Name, ... )
```

Diese Entitätsmenge besitzt eine "ist_ein" - Beziehung zu Hochschulmitarbeiter.

Beziehungsmengen sind

```
Betreut
```
("Professor - Diplomarbeit"). Der Beziehungstyp ist hierarchisch (1 : n)
```
Erarbeitet
```
Der Beziehungstyp ("Student - Vorlesung - Professor") ist vom Typ n : p : m und besitzt die Attribute Semester, Jahr.
```
Schreibt
```

[22] vgl. 1.3.1.2

[23] Identifizierende Attribute sind unterstrichen

[24] Die Mitarbeiternummer ist ein Attribut, das allen Hochschulmitarbeitern gemeinsam ist und veerbt wird

Der Beziehungstyp "Diplomarbeit - Student" ist vom Typ m : n und besitzt das Attribut Semester.

Das führt zu dem folgenden Entity-Relationship-Diagramm:

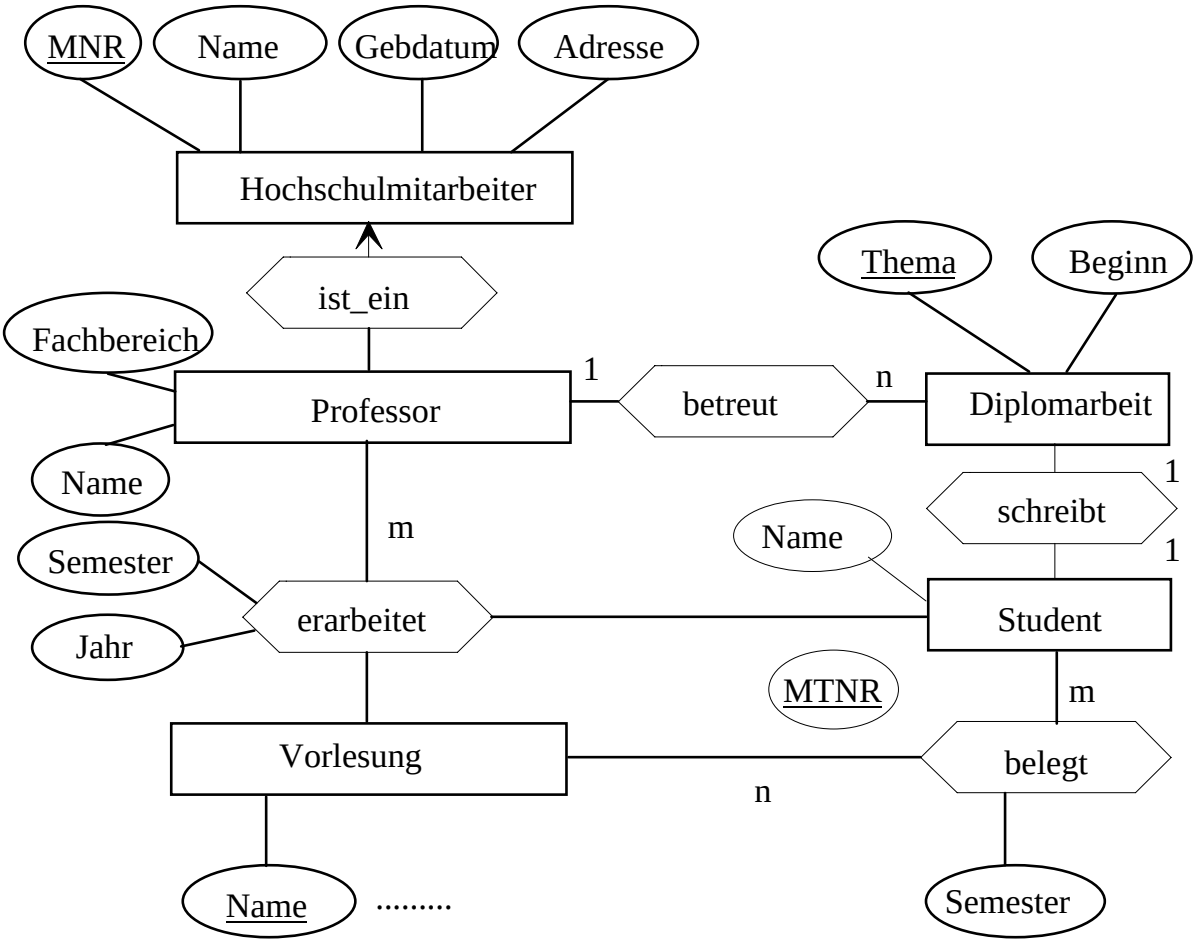
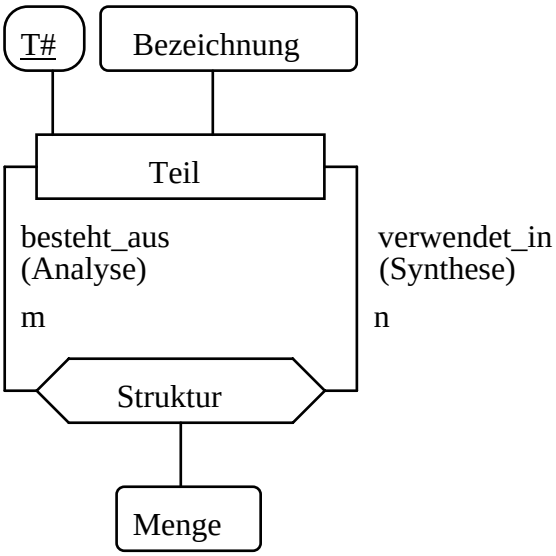


Abb. 1.3-7: Entity-Relationship-Diagramm zu einer Hochschul-Verwaltung

Homogene Zusammenstellungen von Beziehungsmengen der Klassifikation E - E sind ebenfalls im ERM darstellbar, z.B. Stückliste und Verwendungsnachweis bilden sich im ER-Diagramm so ab:



Zur Darstellung der Mengen greift das ERM auf das Relationenmodell zurück.

Teil	
T#	Bezeichnung
E1	Einzelteil-1
E2	Einzelteil_2
E3	Einzelteil_3
B1	Baugruppe_1
B2	Baugruppe_2
P1	Endprodukt_1
P2	Endprodukt_2

Struktur		
OT#	UT#	Menge
P1	B1	2
P1	B2	3
P1	E3	10
P2	B1	3
P2	E3	8
B1	E1	7
B1	E2	8
B2	E2	10
B2	E3	4

Abb. 1.3-8: Entity-Relationship-Diagramm für eine Teile-Verwaltung

"**Dreierbeziehungen**"

Die Beziehungsmenge "Student-Vorlesung-Professor (n:p:m) ist eine Dreierbeziehung (Dreifachbeziehung). Auch 1:n:m-, 1:1:n- und 1:1:1-Beziehungen sind möglich.

Bsp.: Studenten an Fachhochschulen haben zwei praktische Studiensemester zu absolvieren. Es wird gefordert, daß im Rahmen des Praktikums eine Mitwirkung des Studenten an Projektarbeiten stattfindet. Zweckmäßigerweise gilt dann:

- kein Betreuer führt einen beliebigen Studenten in mehr als einem Projekt
- kein Student arbeitet an einem beliebigen Projekt, das von mehr als einem Betreuer betreut wird.

Das ER-Modell drückt das so aus:

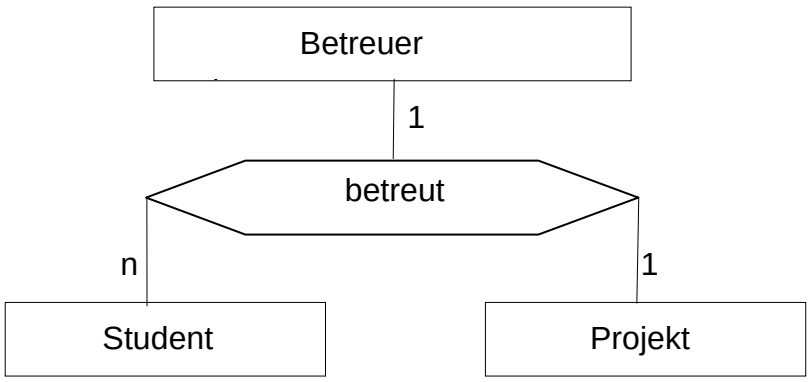


Abb. 1.3-9: Eine 1:1:n-Dreierbeziehung

„**Konstruktionsoperatoren**“

Durch Generalisierung werden ähnliche oder miteinander verwandte Objekttypen zu übergeordneten Objekttypen zusammengefaßt. In einem ER-Diagramm kann die Generalisierung durch eine „ist_ein“-Beziehung veranschaulicht werden.

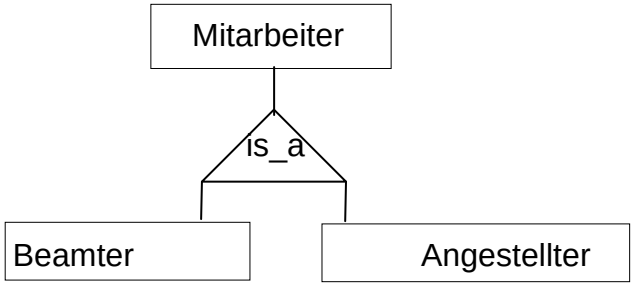


Abb. 1.3-10: ER-Diagramm zur Beschreibung der Mitarbeiter an einer FH

Spezialisierung zerlegt Objekttypen in speziell definierte nachgeordnete Entitätstypen. Spezialisierung ist die Umkehrung der Generalisierung. So ist die vorstehende Abbildung auch eine Spezialisierung, da dier Entitätstyp „Mitarbeiter“ in die Entitätstypen „Beamter“ und „Angestellter“ spezialisiert wird.
Mit der "**ist_ein**"-Beziehung (is_a) werden **Untertypen** (Subtypen) in das ERM eingeführt. Ein Entitätstyp E_1 ist ein Untertyp des Entitätstyps E_2, falls jede Ausprägung (Instanz) von E_1 auch ein Untertyp von E_2 ist.

Bsp.: In einem Software-Unternehmen für Prozeßautomatisierung gibt es Sekretärinnen, Manager und Ingenieure. Die Ingenieure können Maschinenbau-, Flugzeugbau-, Bauingenieure sein. Im ER-Diagramm drückt sich das so aus:

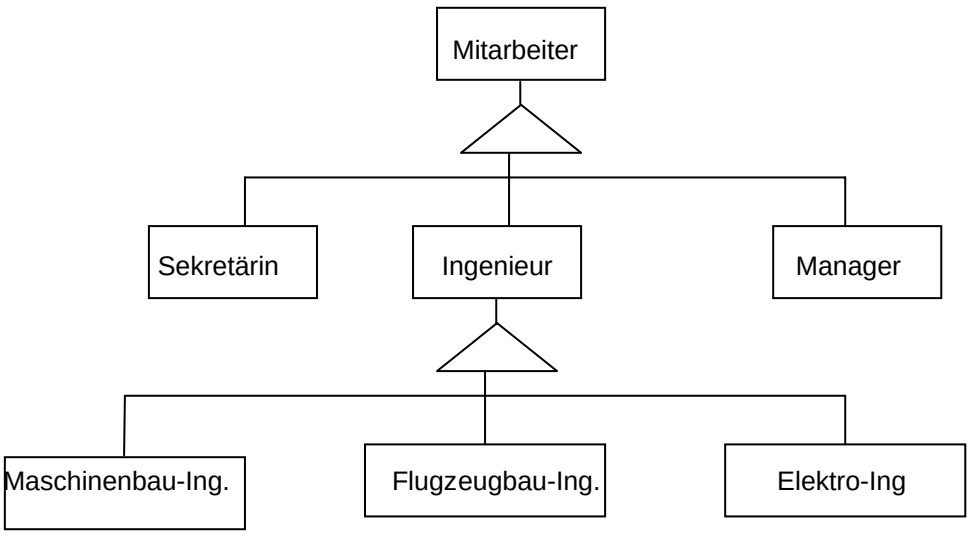


Abb. 1.3-11: Mitarbeiterbeziehungen in einer Projektgruppe

Diese Hierarchie könnte in das folgende Schema einer relationalen Datenbank überführt werden:

Mitarbeiter(MNR, (Attribute, die allen Mitarbeitern gemeinsam sind))
Ingenieure(MNR, (Attribute, spezifisch für Ingenieure))
Sekretärin(MNR, (Attribute, spezifisch für Sekretärinnen, Sekretäre))
Maschinenbau-Ingenieure(MNR, (Attribute, spezifisch für Maschinenbau-Ingenieure))
Flugzeugbau-Ingenieure(MNR, (Attribute, spezifisch für Flugzeugbau-Ingenieure))

Elektro-Ingenieure(MNR, (Attribute, spezifisch für Elektro-Ingenieure))
Manager(MNR, (Attribute, spezisch für Manager))

Entitäten, die Mitglieder eines Untertyps sind, erben die Attribute ihrer übergeordneten Typen. Untertypen könen zusätzliche spezifische Attribute und Beziehungen haben.
Durch Aggregation werden Beziehungen und zugehörige Entitäten zu Entitäten auf höherer Ebene zusammengefaßt. So kann in dem folgenden ER-Diagramm „Kunde“, „Artikel“ und der Beziehungstyp „bestellt“ zusammengefaßt werden.

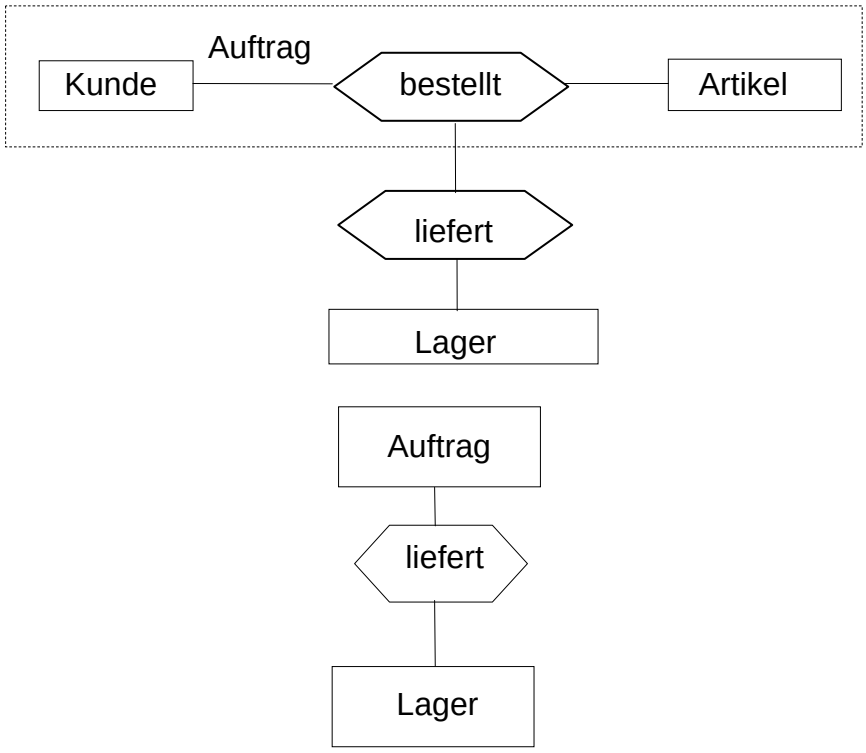


Abb. 1.3-12: Aggregation

Schwache Entitätstypen

Manchmal ist es nicht möglich, eine Entität des Typs E anhand der Ausprägungen seiner Attribute zu identifizieren. In solchen Fällen wird eine Entität E über die Beziehungsmenge E-E' identifiziert. E ist dann ein schwacher Entitätstyp. Jeder Beziehungstyp, der mit einem schwachen Entitätstyp verknüpft ist, ist ein schwacher Beziehungstyp. Im ER-Diagramm wird ein schwacher Entitätstyp durch ein doppelt umrahmtes Rechteck, die Richtung der Abhängigkeit durch einen Pfeil symbolisiert, z.B.:

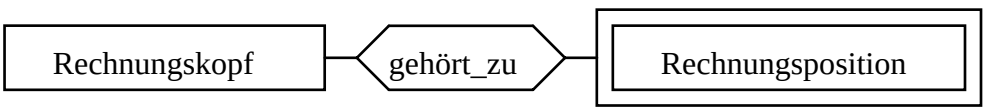


Abb. 1.3-13: Entity-Relationship-Diagramm für eine Rechnung

Durch das Konzept des schwachen Entitäts-Typs werden Existenzabhängigkeiten in das ERM eingeführt. In dem vorliegenden Beispiel hängt die Existenz einer Rechnungsposition von der Existenz des zugehörigen Rechnungskopfes ab. Es

besteht eine Schüsselabhängigkeit zu einer anderen Entität. Beziehungstypen, die diese Identifizierung herbeiführen, werden häufig auch im ER-Diagramm in einer Raute mit doppeltem Umriß gezeichnet.

Umwandlung des ER-Modells in ein relationales Modell

Entitätsmengen: Für jede Entitätsmenge wird eine Relation erstellt, die für jedes Attribut eine Spalte besitzt. Die Primärschlüssel werden übernommen.
1:1- und 1:n-Beziehungen: Eine der beiden Relationen wird um ein Attribut (Spalte) erweitert. Bei 1:1-Beziehungen ist es egal, welche der beiden Relationen erweitert wird. In der 1:n-Beziehung wird die Relation erweitert, in der das „n“ steht. Besitzt die Relation beschreibende Attribute, so werden diese zusätzlich noch bei der erweiterten Relation angegeben. Bei dieser Darstellung können Nullwerte auftreten.
m:n-Beziehungen: Bei einer m:n-Beziehung ist immer eine zusätzliche Relation zur Verknüpfung der Relationen erforderlich, um die betreffenden Entitäten zu verknüpfen. Diese Relation enthält die Primärschlüssel der beiden Relationen und kann zusätzlich noch beschreibende Attribute enthalten.

Bsp.: Gegeben ist das folgende ERM-Diagramm

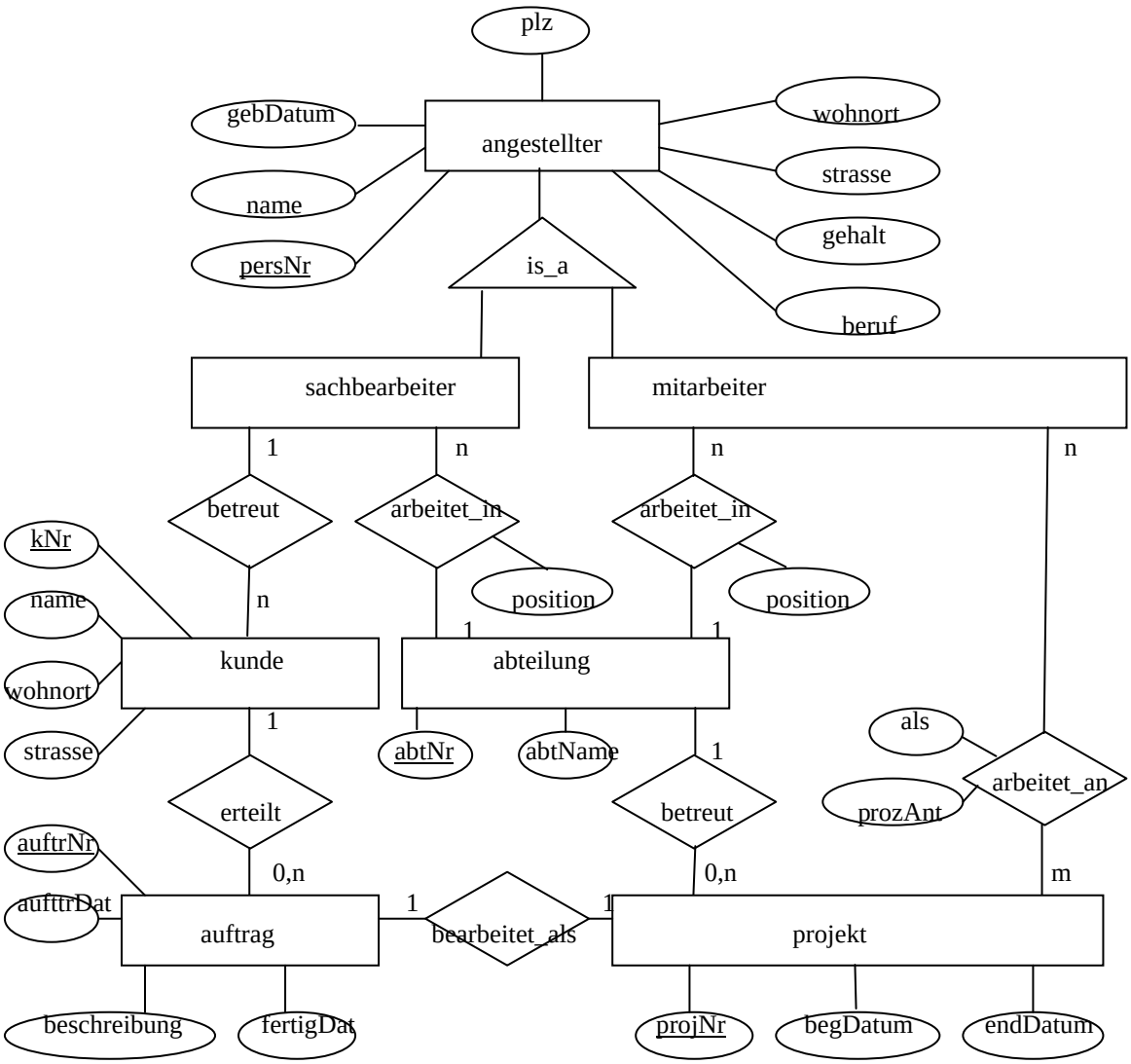


Abb.:

Setze dieses ERM-Diagramm in ein relationales Modell in 3. Normalform um. Die Umsetzung soll durch das Schema der relationalen Datenbank beschrieben werden

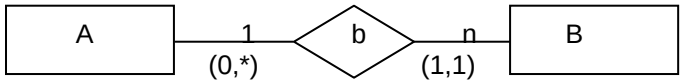
angestellter(persNr,name,gebdatum,plz,wohnort,strasse,gehalt,beruf,position)
kunde(kNr,name,plz,wohnort,strasse)
abteilung(abtnr,abtname)
auftrag(auftrNr,auftrDat,beschreibung,fertigDat,projNr)
projekt(projNr,begDatum,endDatum,abtNr)
arbeitet_an(persNr,projNr,taetigkeit,prozAnt)

Erweiterungen des ERM

Zum ERM wurde eine Vielzahl von Varianten und Erweiterungen vorgeschlagen. **Eine** sehr sinnvolle **Erweiterung** bezieht sich auf die Präzisierung der Komplexität von Beziehungen. Die bisher angegebene Darstellung der Kompexität von Beziehungen ist durch das Verhältnis zwischen 2 Entitäten bestimmt. Eine "1:n"-Beziehung zwischen z.B. einem Kunden und einer Rechnung sagt aber nichts darüber aus, ob jedem Kunden wenigstens eine Rechnung zugeordnet sein muß oder nicht. Auch ist nicht ersichtlich, ob sich die Rechnung genau auf einen Kunden bezieht oder ob Rechnungen ohne Kunden zulässig sind. Bei drei- und mehrstelligen Beziehungen ist die "(1,m,n)"-Schreibweise überhaupt nicht sinnvoll interpretierbar. Wird für jeden Entitätstyp durch einen Komplexitätsgrad comp(E,b) angegeben, mit wievielen Beziehungen des Typs b eine Entität minimal in Beziehung stehen muß bzw. maximal in Beziehung stehen kann, dann liegt die "(min,max)"-Schreibweise vor. Es gilt

0 <= min <= 1 <= max <= * [25]

Eine Beziehung b(A,B) wird durch 2 Komplexitätsgrade comp(A,b) und comp(B,b) beschrieben, z.B.:



Der als Zahlenpaar angegebene Komplexitätsgrad bedeutet:

- Die erste Zahl vor dem Komma gibt die Mindestzahl an
- Die zweite Zahl nach dem Komma gibt die Höchstzahl an. „*“ bedeutet viele.

Eine Beschreibung des Beziehungstyps b(A,B) kann durch die Komplexitätsgrade comb(A,b) bzw. comp(B,b) allgemein so erfolgen:

b(A,B)	comp(A,b)	comp(B,b)
1:1	(0,1) oder (1,1)	(0,1) oder (1,1)
1: n	(0,*) oder (1,*)	(0,1) oder (1,1)
n:1	(0,1) oder (1,1)	(0,* oder (1,*)
n:m	(0,*) oder (1,*)	(0,*) oder (1,*)

Abb. 1.3-14: Komplexitätsgrade im ERM

[25] * bedeutet beliebig viele

Bsp.: Ein typisches ER-Diagramm aus der Fertigungsvorbereitung

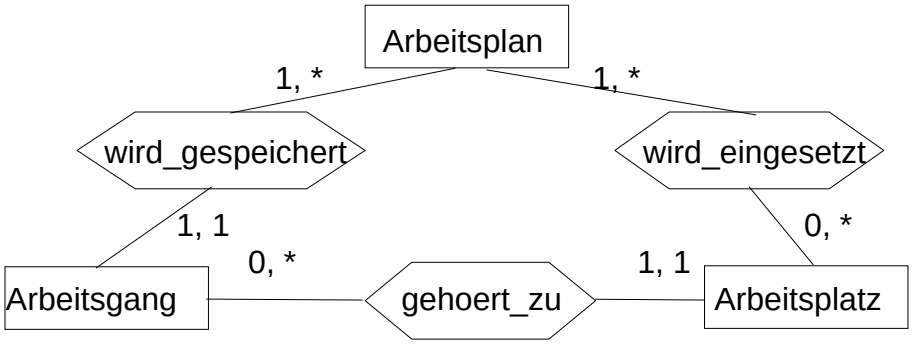


Abb. 1.3-15: ER-Diagramm aus der Fertigungsvorbereitung

Zusammenfassung: „Arbeitsschritte beim Ausarbeiten von Datenstrukturen in einem ER-Diagramm“

(1) Ermittle relevante Entitäten für den betrachteten Datenbereich
(2) Feststellen der Entitätsmengen. Nur gleichartige Entitäten können zur Entitätsmenge zusammengefaßt werden.
(3) Ermitteln der Attribute für jede Entitätsmenge (evtl. mit Korrekturen des 2. Arbeitsschrittes)
(4) Definition der Beziehungstypen von Entitätsmengen, zwischen denen Beziehungen bestehen.
(5) Ermitteln der Häufigkeiten der Beziehungen zwischen den Entitätsmengen
(6) Darstellung der Datenstruktur in einem ER-Diagramm

Aufgabe: „Entwerfe das ER-Diagramm, das Schema und die relationale Datenbank für die Personalverwaltung in einem Unternehmen".

a) Entwurf des ER-Diagramms

Die Analyse zur Ermittlung der Struktur einer relationale Datenbank hat ergeben:

(1) Ermitteln relvanter Entitäten für den betrachteten Datenbereich, z.B.:

- Der Angestellte "A1" mit dem Namen "Fritz" ist am "2.1.1950" geboren und arbeitet in der Abteilung "OD (Organisation und Datenverarbeitung)" als "Systemplaner (SY)". Er besitzt zusätzliche Qualifikationen für "Operateur (OP)-", "Programmierer (PR)"-Tätigkeiten. Ein "Systemplaner" verdient "6000.00.-DM", ein "Operateur (OP)" "3500.-DM", ein "Programmierer" 5000.00.-DM" im Monat.
- Der Angestellte "A2" mit dem Namen "Tom" ist am "2.3.1951" geboren und arbeitet in der Abteilung "KO (Konstruktion)" als "Ingenieur (IN)". Er besitzt zusätzliche Qualifikation zur "Systemplaner (SY)-"Tätigkeiten. Ein "Systemplaner" verdient "6000.00.-DM", ein "Ingenieur (IN)" "6000.- DM" im Monat.
- Der Angestellte "A3" mit dem Namen "Werner" ist am "23.1.1948" geboren und arbeitet in der Abteilung "OD (Organisation und Datenverarbeitung)" als "Programmierer (SY)". Er besitzt zusätzliche Qualifikationen für "Operateur (OP)-", "Programmierer (PR)"-Tätigkeiten. Ein "Programmierer" verdient "5000.00.-DM" im Monat.
- Der Angestellte "A4" mit dem Namen "Gerd" ist am "3.11.1955" geboren und arbeitet in der Abteilung "VT (Vertrieb)" als "Kaufm. Angestellter (KA)". Ein "Kaufm. Angestellter" verdient "3000.00.-DM" im Monat.
- Der Angestellte "A5" mit dem Namen "Emil" ist am "2.3.1960" geboren und arbeitet in der Abteilung "PA (Personalabteilung)" als "Programmierer (PR)". Ein "Programmierer" verdient "5000.00.-DM" im Monat.

- Der Angestellte "A6" mit dem Namen "Uwe" ist am "3.4.1952" geboren und arbeitet in der Abteilung "RZ (Rechenzentrum)" als "Operateur (OP)". Ein "Operateur (OP)" verdient "3500.- DM", ein "Programmierer" im Monat.
- Die Angestellte "A7" mit dem Namen "Erna" ist am "17.11.1955" geboren und arbeitet in der Abteilung "KO (Konstruktion)" als "Techn. Angestellte (TA)". Eine "Techn. Angestellte" verdient "3000.00.-DM" im Monat.
- Die Angestellte "A8" mit dem Namen "Rita" ist am "2.12.1957" geboren und arbeitet in der Abteilung "KO (Konstruktion)" als "Techn. Angestellte (TA)". Eine "Techn. Angestellte" verdient "3000.00.-DM" im Monat.
- Die Angestellte "A9" mit dem Namen "Ute" ist am "8.9.1962" geboren und arbeitet in der Abteilung "OD (Organisation und Datenverarbeitung)" als "Systemplaner (SY)". Sie besitzt zusätzliche Qualifikation für "Ingenieur (IN)"-Tätigkeiten. Ein "Systemplaner" verdient "6000.00.-DM", ein "Ingenieur (IN)" "6000.- DM" im Monat.
- Der Angestellte "A10" mit dem Namen "Willi" ist am "7.6.1956" geboren und arbeitet in der Abteilung "KO (Konstruktion)" als "Ingenieur (IN)". Ein "Ingenieur" verdient "6000.00.-DM" im Monat.
- .....

(2) Daraus lassen sich die folgendenen Entitäten ableiten

- Abteilung, Angestellte, Job

(3) , die folgende Attribute besitzen:

- Abteilung(Abt_ID,Bezeichnung)
- Angestellte(Ang_ID,Name,Gebjahr,Abt_ID,Job_ID)
- Job(Job_ID,Titel,Gehalt)

(4) Beziehungen bestehen zwischen

Abteilung und Angestellte
Angestellte und Job (ausgeübeter Beruf)
Angestellte und Job (Qualifikation)

(5) Ermitteln der Häufigkeit von Entitätsmengen zwischen denen Beziehungen bestehen:

Ein Angestellter arbeitet in einer Abteilung.
Eine Abteilung hat mehrere Angestellte.
Ein Angestellter übt eine bestimmte berufliche Tätigkeit aus.
Eine berufliche Tätigkeit kann von mehreren Angestellten wahr genommen werden.
Ein Angestellter besitzt mehrere Qualifikationen,
Eine bestimmte Qualifikation haben mehrere Angestellte erreicht.

(6) Darstellung der Datenstruktur in einem ER-Diagramm

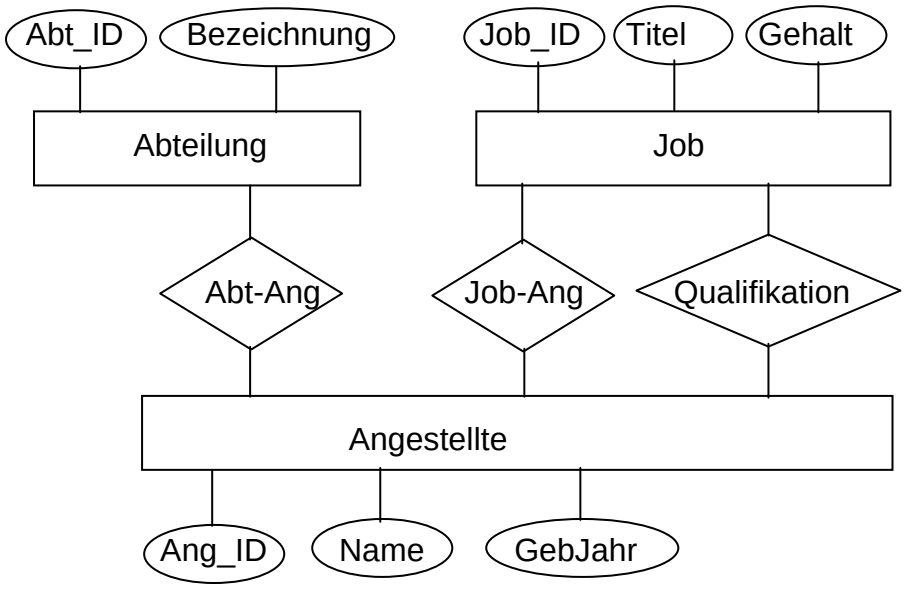


Abb. 1.3-16: ER-Diagramm zur Datenbank Personalverwaltung

b) Schemaentwurf der relationalen Datenbank

- `Abteilung(Abt_ID,Bezeichnung)`
- `Angestellte(Ang_ID,Name,Gebjahr,Abt_ID,Job_ID)`
- `Job(Job_ID,Titel,Gehalt)`
- `Qualifikation(Ang_ID,Job_ID)`

c) Die relationale Datenbank für das Personalwesen

ABTEILUNG

ABT_ID	BEZEICHNUNG
KO	Konstruktion
OD	Organisation und Datenverarbeitung
PA	Personalabteilung
RZ	Rechenzentrum
VT	Vertrieb

ANGESTELLTE

ANG_ID	NAME	GEBJAHR	ABT_ID	JOB_ID
A1	Fritz	2.1.1950	OD	SY
A2	Tom	2.3.1951	KO	IN
A3	Werner	23.4.1948	OD	PR
A4	Gerd	3.11.1950	VT	KA
A5	Emil	2.3.1960	PA	PR
A6	Uwe	3.4.1952	RZ	OP
A7	Eva	17.11.1955	KO	TA
A8	Rita	02.12.1957	KO	TA
A9	Ute	08.09.1962	OD	SY
A10	Willi	7.7.1956	KO	IN
A11	Erna	13.10.1966	OD	KA
A12	Anton	5.7.1948	OD	SY
A13	Josef	2.8.1952	KO	SY
A14	Maria	17.09.1964	PA	KA

JOB

JOB_ID	TITEL	GEHALT
KA	Kaufm. Angestellter	3000,00 DM
TA	Techn. Angestellter	3000,00 DM
SY	Systemplaner	6000,00 DM
PR	Programmierer	5000,00 DM
OP	Operateur	3500,00 DM

QUALIFIKATION

ANG_ID	JOB_ID
A1	SY
A1	PR
A1	OP
A2	IN
A2	SY
A3	PR
A4	KA
A5	PR
A6	OP
A7	TA
A8	IN
A9	SY
A10	IN
A11	KA
A12	SY
A13	IN
A14	KA

Abb. 1.3-17: Tabellen zur relationalen Datenbank

1.3.4 Legacy-Datenbanken

1.3.4.1 Das netzwerkorientierte Datenbankmodell

Das Datenmodell einer netzwerkorientierten DB sieht vor, daß die gespeicherten Objekte beliebig miteinander verknüpft sein können. Durch dieses Datenmodell werden Objekttypen und die vielfältigen Beziehungen, die zwischen Objekttypen bestehen können, beschrieben.

1. Beziehungen

Die Beziehungen zwischen den Objekttypen haben die Form einfacher Netzwerkstrukturen und tragen eine Benennung. In einer grafischen Darstellung sind das Pfeile, die die in eckigen Kästchen dargestellten Objekttypen (Entitäts-, Satztypen) verbinden. Die Benennung der Beziehungstypen (Set-Typen) nimmt ein Oval (Ellipse) auf. Eine Beziehung, z.B. zwischen den Entitätstypen E bzw. E' stellt sich dann folgendermaßen dar:

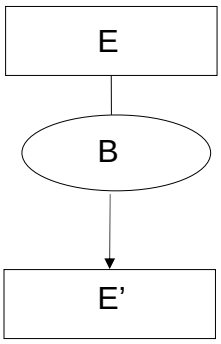


Abb. 1.3-20: Datenbankstrukturdiagramm im Netzwerkmodell

Beschreibt man in dieser Form einen umfassenden Bereich (z.B. alle Daten eines Unternehmens), so erhält man einen gerichteten Graphen:

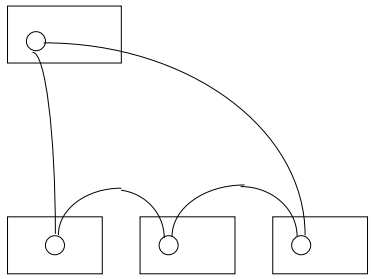
- Seine Knoten sind Entitätstypen
- Seine Kanten sind Beziehungen zwischen den Entitätstypen, da alle möglichen Beziehungen zugelassen sind, erhält man ein Netzwerk (plex structure).

2. Definition des Set-Typ

Jede Entität e' vom Typ E' steht höchstens mit einer Entität e vom Typ E in Bezie-hung.
Zu jeder Entität e vom Typ E gehört ein Set vom Set-Typ B. Neben e gehören dazu alle Entitäten e_1',e_2', ... , e_k' vom Typ E'.
Ein Set-Typ bestimmt eine "1 : n"-Beziehung.
Die Ausprägungen eines speziellen Set-Typs beschreibt man mit Hilfe eines Occurrence-Diagramms:

Nichtleerer Set

Owner



Member

Abb. 1.3-21: Occurrence-Diagramm im Netzwerkmodell

Leerer Set: Der Owner besitzt keine Member

3. Die Datenbank des Netzwerk-Datenmodell

- Es gibt eine Menge von benannten Entitäts-Typen und eine Menge von benannten Set-Typen
- Jede Entität der Datenbank gehört genau zu einem Entitäts-Typ, jede konkrete Beziehung zwischen Entitäten zu genau einem Set-Typ
- Den Set-Typen sind keine Attribute zugeordnet
- Die Member jedes Set sind (gemäß einer dem Set-Typ zugeordneten Ordnung) geordnet.

- Eine Teilmenge von Entitäts-Typen ist ausgezeichnet. Für sie gilt: Zu jeder Entität e der Datenbank gibt es mindestens einen (gerichteten) Weg $e_0,e_1,e_2,....,e_{k-1},e_k$,dessen Ausgangspunkt e_0 von einem Typ E_0 ist. Entitäten von einem Typ E_0 heißen Einstiegpunkte in das Netzwerk.
- Jede Entität, die nicht von einem Typ E_0 ist, kann nur über eine vom Einstieg-punkt ausgehende Folge von Sets erreicht und dem Benutzer zugänglich gemacht werden. Dabei kann ein Set auch vom Member zum Owner durchlaufen werden.

Bsp.: Komplexe Beziehungen zwischen Angestellten und Projekten



Abb. 1.3-22: ER-Diagramm für "Angestellter" und "Projekt"

"A_P" ist hier vom Typ m:n, denn

- ein Angestellter arbeitet an mehreren Projekten bzw.
- ein Projekt wird von mehr als einem Angestellten bearbeitet

z.B.: Es bestehen zwischen den 4 Angestellten A1, A2, A3, A4 und den Projekten P1, P2, P3 die folgenden konkreten Beziehungen:

	P1	P2	P3
A1	x	x	
A2		x	
A3		x	
A4	x		x

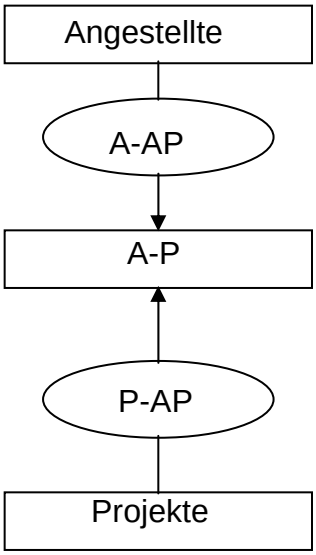
Realisierung im Netzwerk-Modell?

Notwendig ist die Einführung eines neuen, der Beziehung A_P entsprechenden Entitäts-Typ E_B (z.B. A-P). Zu jeder konkret auftretenden Beziehung wird festgelegt:

- Eine Entität (Ai-Pj) vom Typ E_B
- 2 Set-Typen b_1 (A-AP) und b_2 (P-AP)

Die neue Entität A-P kann jetzt auch Attribute zugeordnet bekommen (,z.B.: % der Arbeitszeit)

Datenbankstruktur-Diagramm



Occurrence-Diagramm

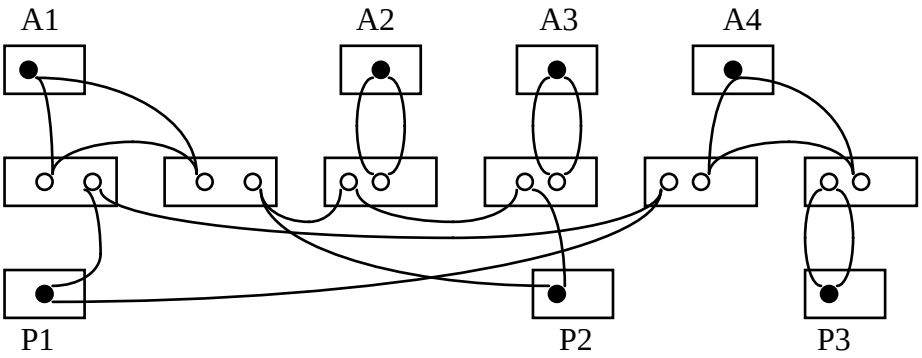


Abb. 1.3-23: Datenbankstruktur- und Occurrence-Diagramm zur Beziehung "Angestellter" und "Projekt"

4. Zugriff auf eine Entität

Von einem Einstiegpunkt in die Datenbank gibt man einen genauen Weg in der Form einer Folge von Sets an, über die man die gewünschte Entität erreichen kann. Der Benutzer muß den Zugriffspfad für jede Entität selbst angeben (Navigieren durch die Datenbank).

Bsp.: Zugriff zu den Angestellten (im Rahmen der komplexen Beziehungen zwischen Angestellten und Projekten)

Aufgabe: Finde alle Angestellte, die an Projekt P1 arbeiten!

Lösungsschritte:

1. Suche Entität P1
2. Deute P1 als Owner eines Set vom Typ P-AP
3. Suche 1./nächstes Member Ai-P1 in diesem Set
4. Falls nicht vorhanden: STOP (evtl. weiter mit 8.)
5. (Vorhanden). Deute Ai-P als Member in einem Set vom Typ A-AP, suche zugehörigen Owner Ai
6. Verarbeite Ai
7. Weiter bei 3.
8. ...

5. Zusammenfassung

Die Einführung spezieller Entitäts-Typen zur Realisierung von Beziehungen ist in folgenden Fällen notwendig:

1. Die Beziehungen sind keinem Set-Typ zuzuordnen ("n:m"-Beziehungen)
2. Die Beziehung besteht zwischen mehr als 2 Entitäts-Typen
3. Eine Beziehung besitzt Attribute

Das Schema einer auf dem Netzwerk-Datenmodell beruhenden Datenbank muß also enthalten:

- Die Beschreibung der vorhandenen Entitäts-Typen und ihrer Attribute (einschl. der Wertebereiche)
- Die Beschreibung der vorkommenden Set-Typen

Bsp.:

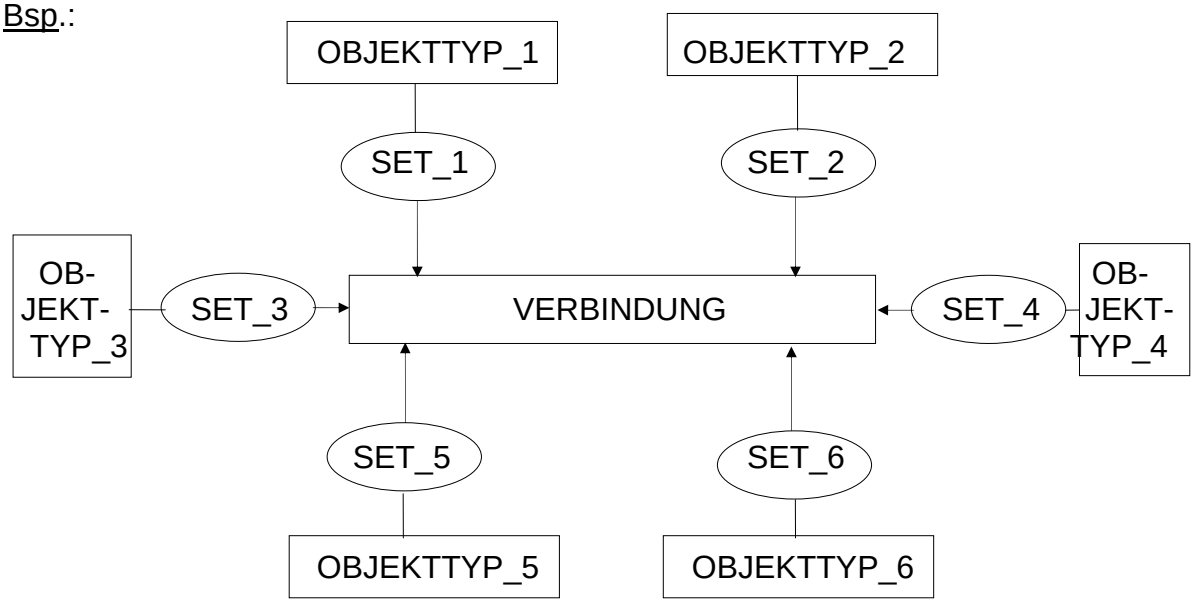


Abb. 1.3-24: Datenbankstrukturdiagramm eines Netwerks mit 6 Sets

1.3.4.2 Das hierarchische Datenbankmodell

Berücksichtigt werden hierarchische Beziehungen zwischen den Daten. Eine hierarchische Struktur läßt sich in der Gestalt eines Baumes beschreiben. Ein Baum ist eine Datenstruktur mit folgenden Eigenschaften:

- Sie ist zyklenfrei.
- Es gibt einen speziell hervorgehobenen Knoten, die Wurzel. Sie hat keinen Vorgänger.
- Auf jeden anderen Knoten zeigt eine Kante, d.h.: Jeder Knoten hat genau einen Vorgänger.

1. Definition

- Es gibt eine Menge benannter Entitäts-Typen (Segmente) und eine Menge von unbenannten Beziehungen.
- Jede Entität (Segment Occurrence) gehört zu einem Entitäts-Typ.
- Alle Entitäten innerhalb der Datenbankstruktur (,der zugehörige Graf ist eine Ansammlung von Bäumen,) sind total geordnet (hierarchische Ordnung)
- Jede Entität ist nur über einen Einstiegpunkt (Wurzelsegment) erreichbar.

Bsp.:

Datenbankstruktur-Diagramm

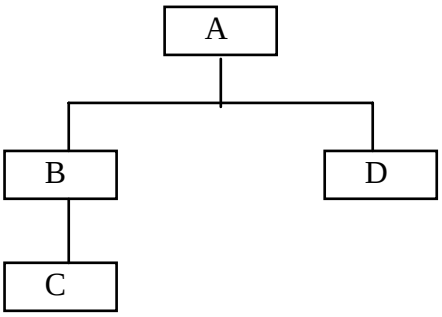


Abb. 1.3-25: Datenbankstruktur im hierarchischen Datenmodell

Occurrence-Diagramm

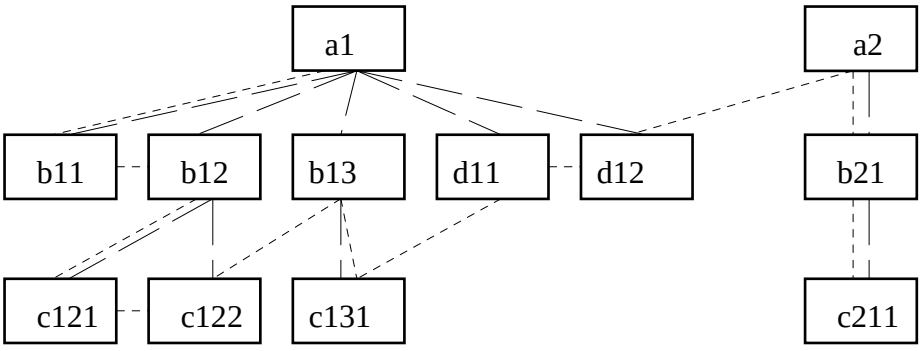


Abb. 1.3-26: Ausprägung hierarchischer Strukturen

2. Wesentliche Einschränkungen gegenüber dem Netzwerkmodell

- Nur Baumstrukturen, keine allgemeine Netzstrukturen sind erlaubt
- Zugriff kann nur über einen gerichteten Weg, der von der Wurzel ausgeht, erfolgen. Nur in einem Baum kann ein Entitäts-Typ vorkommen Ist bspw. die Beziehung E - E' in einem Baum realisiert, ist e' nur über die zugehörige Entität e vom Typ E zu erreichen. Eine selbstständige Verarbeitung von Entitäten des Typs E' ist nur so möglich: Duplizieren aller zu diesem Entitäts-Typ zugehörigen Entitäten (Redundanz!)
- Beziehungen zwischen Entitäts-Typen (Segmenten) sind nur in einer Baum-struktur zu realisieren. Beziehungstypen gibt es daher nur in der Form `1 : 1` bzw. `1 : m`.

Segmente der obersten Hierarchiestufe heißen **Wurzelsegmente**. Ein von einem solchen Segment ausgehende Hierarchie ist ein Datenbanksatz. Die Menge der von den Wurzelsegmenten ausgehenden Sätze bilden eine Datenbasis. Ein Segment ist nur über einen gerichteten Weg erreichbar, dessen Ausgangspunkt das Wurzelsegment ist.

3. Beispiele zur Anwendung hierarchischer Strukturen

a) heterogene Strukturen (`E - E'`)

Bsp.:

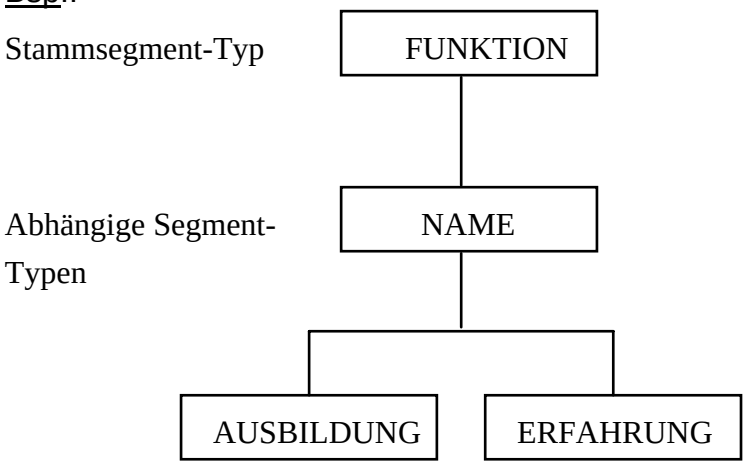


Abb. 1.3-26: Datenbankstrukturdiagramm für eine heterogene Struktur

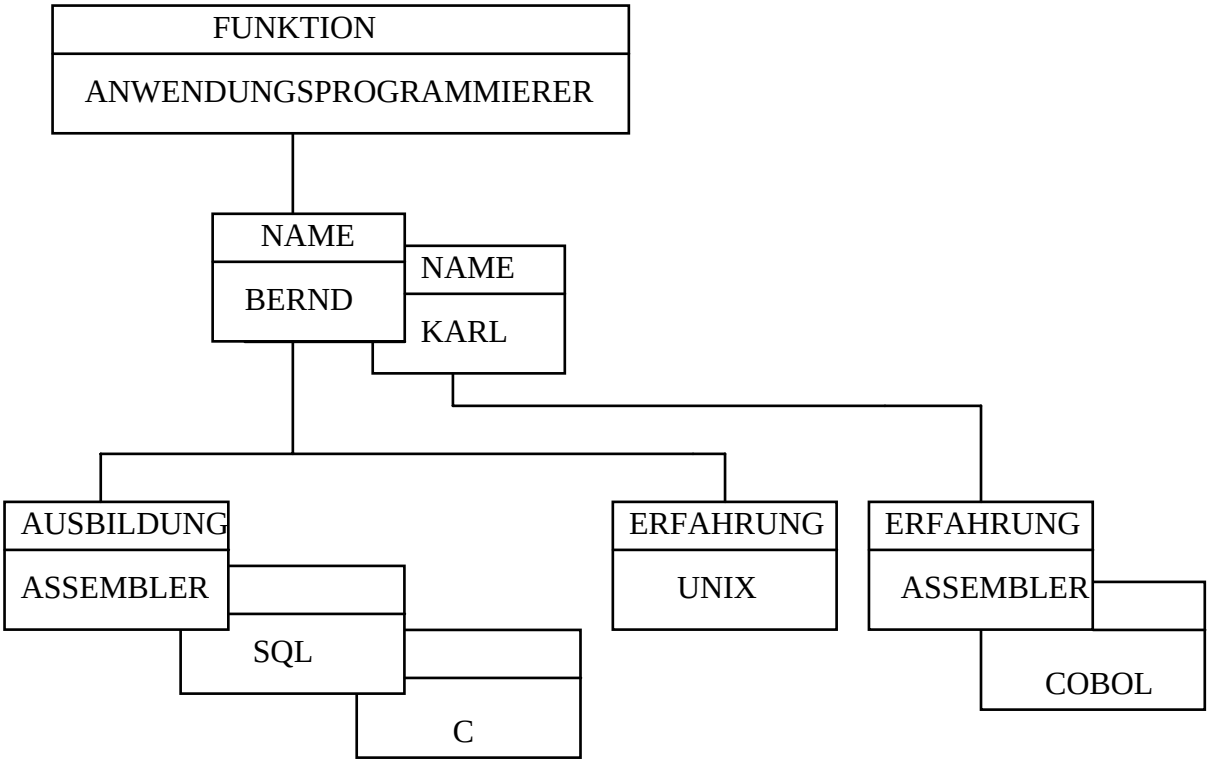


Abb. 1.3-27: Occurrence-Diagramm einer heterogenen Struktur

b) Homogene Strukturen (E - E)

Bsp.:

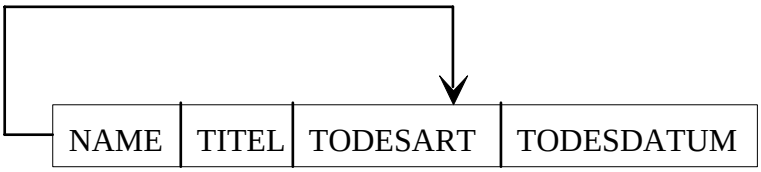


Abb. 1.3-28: Struktur einer Herschaftsfolge

Diese Schemadarstellung beschreibt bspw. die Folge deutscher Könige im Mittelalter. Die Herrscher wurden gewählt, häufig gab es sogar Gegenkönige (1:n Beziehung der Nachfolge). In einer physischen Datenbank des hierarchischen Datenbankmodells sind aber nur 1:1 - bzw. 1:n - Verbindungen zu realisieren. Es muß also zu der vorliegenden Struktur eine Ergänzung (, ein "Zeiger-Segment") geben, damit die Forderung nach einer 1:n Verbindung erfüllt werden kann. Das Zeigersegment ist mit der Zwischensatzart im Netzwerk-Datenbankmodell vergleichbar.

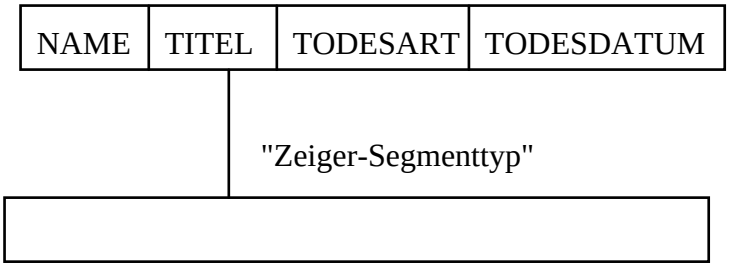


Abb. 1.3-29: Datenbankstrukturdiagramm zur Darstellung einer Herrschaftsfolge

4. Pfadabhängigkeiten

Die auf tieferen Ebenen vorkommenden Segmente sind ohne ihre Vorgänger-Segmente unvollständig:

Bsp.: Gegeben ist folgende Datenbankstruktur:

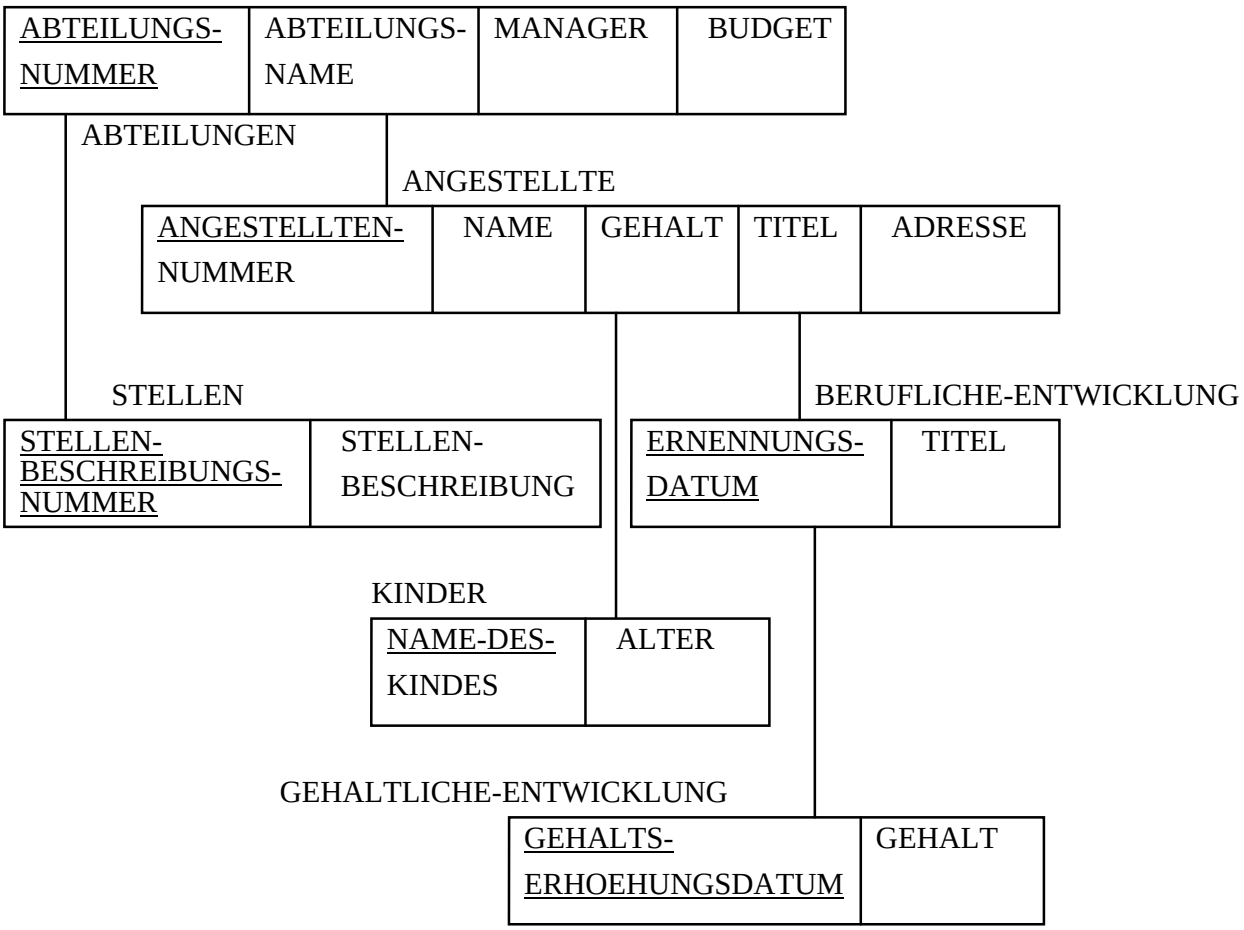


Abb. 1.3-30: Datenbankstrukturdiagramm mit Pfadabhängigkeiten

- Das Segment BERUFLICHE-ENTWICKLUNG ist ohne Bezug auf das Segment ANGESTELLTE bedeutungslos.
- Das Segment GEHALTLICHE-ENTWICKLUNG ist nicht nur vom Vatersegment sondern sogar vom Großvater-Segment ANGESTELLTE abhängig. Eine Schlüsselkombination von ANGESTELLTER und ERNENNUNGSDATUM ist zur Identifizierung des Feldes GEHALT notwendig. Die Schlüsselkombination ABTEILUNGS-NUMMER und STELLEN-BESCHREIBUNGS-NUMMER identifiziert die STELLENBESCHREIBUNG

5. Verbindungen bei hierarchischen Datenbanken (IMS-DL/1)

Sie werden durch Zeiger realisiert. Das Segment, auf das verwiesen wird, heißt "logisches Kind (**logical child**)". Es ist i.a. kein Träger von Daten, sondern dient als Zeigersegment (auf den "logischen Vater (**logical parent**)"). Ein "**logical child**" kann nur als abhängiges Segment auftreten, das zur Vermeidung redundanter Speicherung dient. Logische Elternsegmente können mehrere logische Kinder besitzen. Ein logisches Elternsegment kann aber nicht zugleich auch "logisches Kind" eines "logischen parent" sein.

Zugriff zu Zeigersegmenten

Der Zugriff zu einem Zeigersegment kann über 2 verschiedene Wege erfolgen:

1. Der physische "parent" ist Ausgangspunkt der Abfrage
2. Der logische "parent" ist Ausgangspunkt der Abfrage

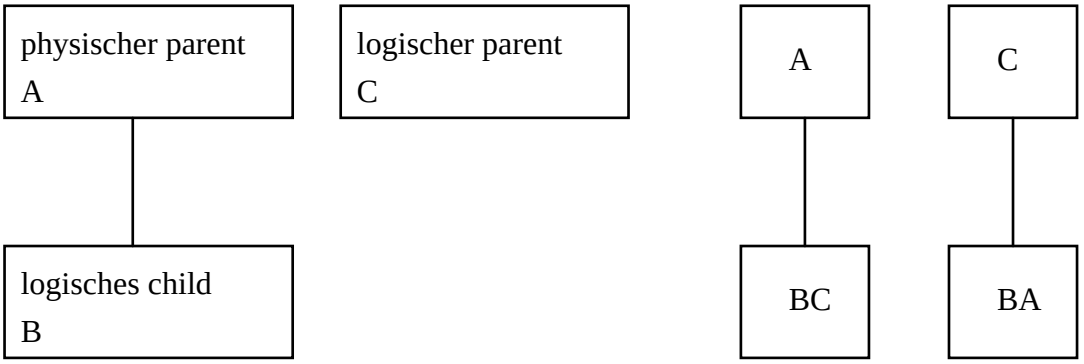


Abb.1.3-31 : Physischer und logischer Parent in hierarchischen Datenbanken

Dem Anwender werden also nicht nur die Daten des log. "parent" zur Verfügung gestellt, wenn er das "log. child" abfragt, sondern eine Kombination (concatenated segment) der Daten des "log. Kindes" und des Segments, das nicht als Ausgangspunkt der Anfrage diente.

6. Logische Datenbankstrukturen

Ausgehend vom "Root-Segmenttyp" einer (logisch verknüpften) phy-sischen Datenbank kann jedes Segment unverändert übernommen werden, mit Ausnahme der "logical child" -Segmente. Statt dessen wird ein neuer Segmenttyp definiert, der aus der Verkettung vom "logical child" und zugehörigem "logical parent" besteht (Übergang zur logisch verknüpften Datenbank). Die dem "logical parent"-Segment über- / untergeordneten Segmente können in der logischen Datenbank als abhängige Segmente des "logical parent" definiert werden. Dabei wird die physische Datenstruktur teilweise invertiert.

1.3.5 Das Koexistenz-Modell

Datenbankarchitekturen bieten im Rahmen des sog. Koexistenzmodells drei Ebenen (Plattformen) für die Beschreibung von Daten an:

1. **Die Ebene der logischen Gestaltung**
   (Dateikonzept des jeweiligen Programmierers, externes Schema)
   Die in Anwenderprogrammen benötigten Daten werden hier (in einem Subschema) beschrieben und bereitgestellt.
2. **Die Ebene der funktionalen Gestaltung**
   Beschrieben wird hier das allumfassende Konzept für die Anwendung der Daten aus der Datenbank (konzeptuelles Schema)
3. **Die Ebene der physischen Gestaltung**
   Beschrieben wird hier die zweckmäßige Ablage der Daten auf peripheren Speichern (internes Schema).

Bsp.: „Beschreibung der Zusammenhänge bei der maschinellen Abwicklung des Bestellwesens“

- Ausgangspunkt ist die BESTELLUNG
  (Bestell-Nr.,Lieferanten-Nr.,Bestell-Datum, Liefertermin,
  Preis (der bestellten Waren (Summe))
- Der Bestellsatz ist unvollständig. Er ist zu ergänzen durch Angabe von Bestell-positionen (Wiederholungsgruppen-Problem!).
  (BESTELLPOSITION)
- Weiterhin möchte man mehr über den Lieferanten wissen!
  (LIEFERANT)
  (Lieferant-Nr.,Name (des Lieferanten),Adresse (des Lieferanten),Information (über den Lieferanten))
- Der Lieferant kann bestimmte Waren (repräsentiert durch eine Teile-Nr.) zu be-stimmten Konditionen (Preis, Liefertermin etc.) liefern.
  (PREISNOTIERUNG)
  (Teile-Nr.,Einzelpreis,Liefertermin, ...)
- Das Teil ist noch zu beschreiben (TEILE)
  (Teile-Nr.,Beschreibung, Vorrat (an Teilen im Lager)
- Schließlich ist der Teilestammsatz noch mit den vorhandenen Bestellungen (Bestell-Nr) zu verknüpfen. Es muß bekannt sein, welche Bestellungen zu einem Teil bereits vorliegen, bei dem der Vorrat zur Neige geht)

Schema

Der folgende Schema-Entwurf bietet sich an

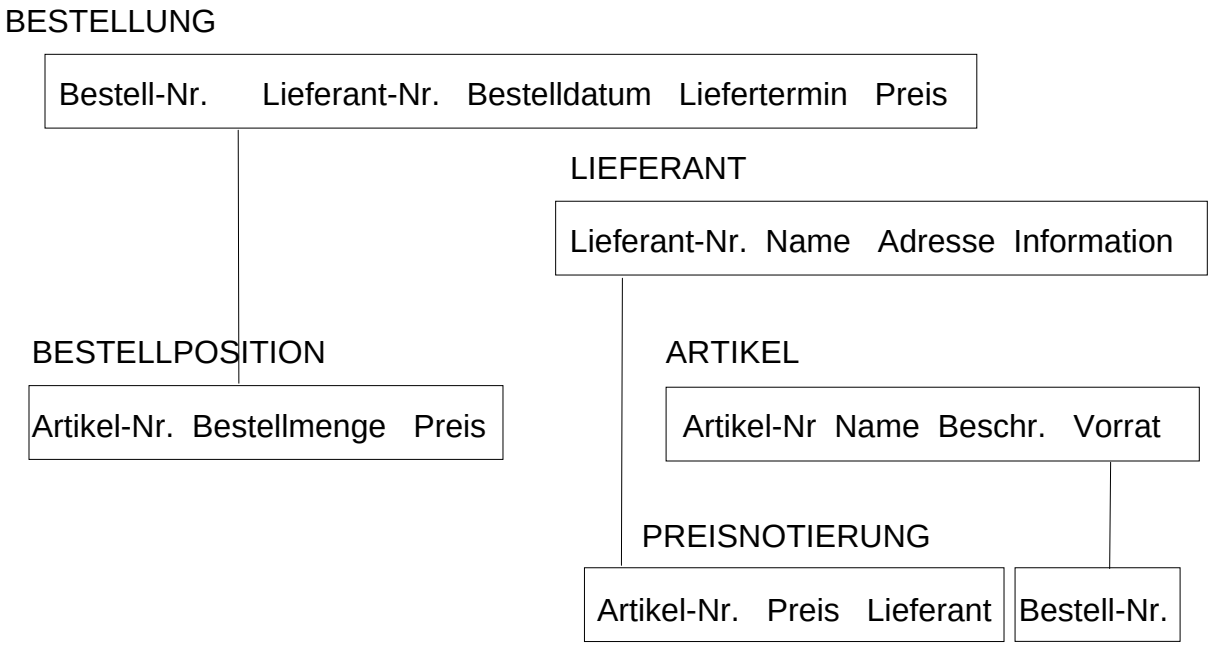


Abb. 1.3-32 : Schema zum maschinellen Abwicklung des Bestellwesens

Der Schema-Entwurf resultiert aus den unmittelbaren Beziehungen zwischen den Objekt- bzw. Satztypen. Außerdem gibt es noch Querbeziehungen. Es handelt sich dabei um ergänzende, nicht unbedingt für den unmittelbaren Verwendungszweck notwendige Informationen.

Subschema

Das Schema ist die allumfassende Darstellung der Datenelemente und Satztypen, die in der DB gespeichert sind. Der jeweilige Anwender braucht davon nur einen Teil, ein vom Schema abgeleitetes Subschema (Sicht, **view**).

Bsp.:

1. Anwendungsprogrammierer A hat die Aufgabe "Maschinelle Abwicklung des Bestellwesens"
A braucht dazu Sicht:

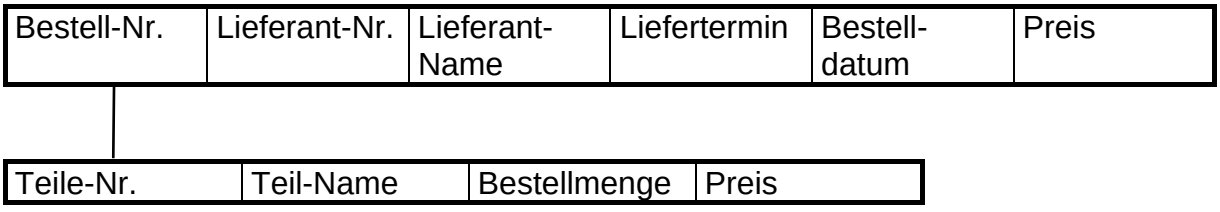


Abb. 1.3-33: Sicht des Anwendungsprogrammierers A

2. Anwendungsprogrammierer B ist für die maschinelle Abwicklung der Nachbestellungen aus dem Lagerwesen zuständig
B braucht dazu die Sicht:

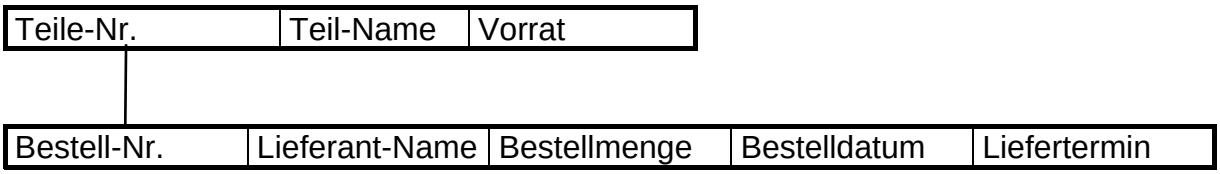


Abb. 1.3-34: Sicht des Anwendungsprogrammiereres B

Das konzeptuelle Schema soll eine vollständige redundanzfreie und neutrale Darstellung der logischen Datenstrukturen gewährleisten. Bei vollständiger Datenneutralität ist es möglich durch unterschiedliche Benutzersichten (Netze, Hierarchien, Relationen) das Basisschema zu betrachten. Eine Sicht wird dabei mit Hilfe einer Datenmanipulationssprache aus der Basis erzeugt.
Auf der anderen Seite soll eine Reihe von Abbildungen verschiedene "Interne Schemata" zur Beschreibung der physischen Strukturen erzeugen. Sie enthalten mit ihren Katalogen die Details zu Zugriffspfaden und zur physischen Speicherung. Eine allgemeine Beschreibung der Schnittstellen enthält der ANSI/SPARC-Architekturvorschlag [26].

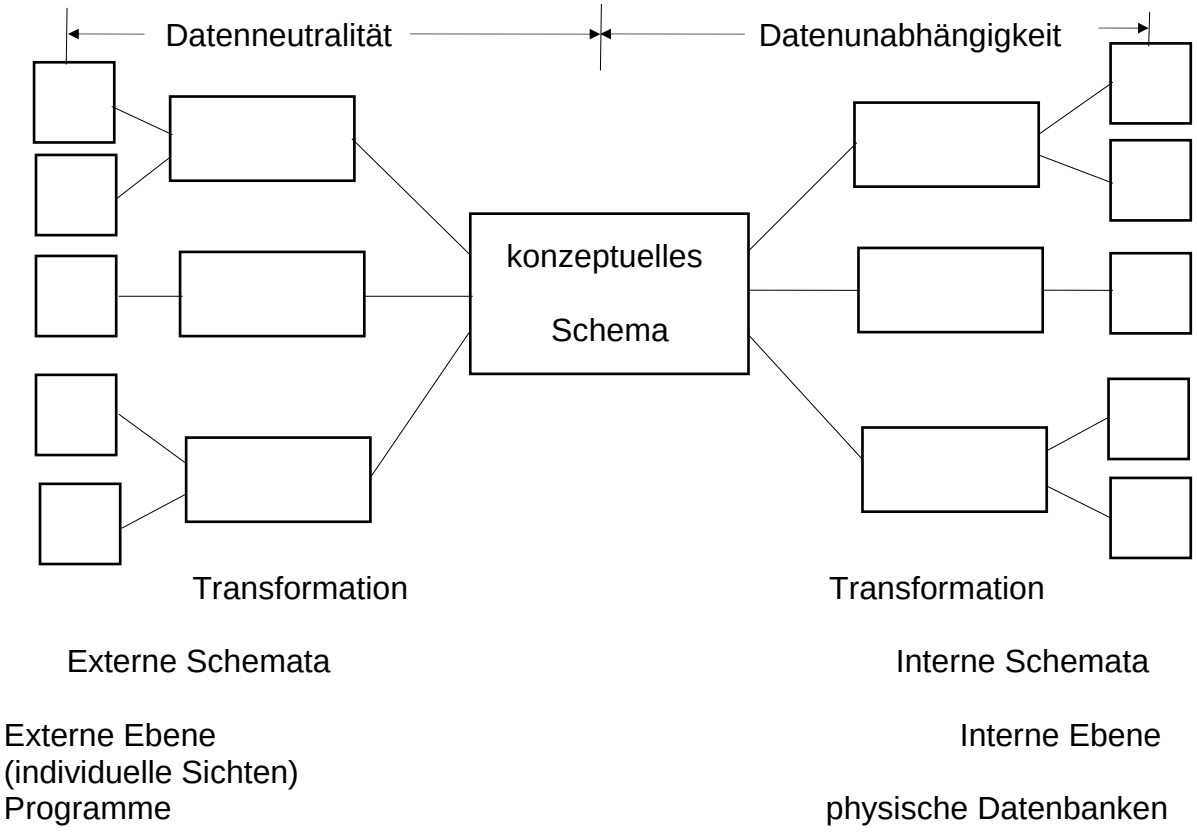


Abb. 1-3-35: Datenbankarchitektur nach ANSI/SPARC

Die **ANSI/SPARC**-Architektur sieht 3 Ebenen vor:

1. Eine mittlere **konzeptuelle Ebene**, die alle drei Anwendersichten zu einer Art gemeinschaftlichen Sicht vereinigt

[26] ANSI/X3/SPARC Group on Data Base Management Systems: Interim Report. Bulletin of ACM SIGMOD, No. 2, (1975)
ANSI(American National Standard Institute)/Sparc(Standard Planning and Requirement Committee

2. Eine **interne Ebene**, die es gestattet, unter Kenntnis von Anwenderprofilen und verfügbarer Hard- und Grundsoftware die leistungsfähigsten Speicher- und Zugriffsmethoden zu wählen
3. Eine **externe Ebene** mit beliebig vielen Anwendersichten

Im Mittelpunkt der verschiedenen Betrachtungen (Sichten) steht das allumfassende Konzept für die Anwendung der Daten. Datenbanken stellen auf der einen Seite die Mittel zur Verfügung, die die jeweiligen Dateien für die spezifischen Anwendungen bereitstellen. Auf der anderen Seite sorgen sie für die Speicherung der Daten (internes Schema, Speicherschema). Entscheidend ist in beiden Fällen das Datenmodell, das der allgemeinen Beschreibung zugrundeliegt.
Das **Datenmodell** soll möglichst genau die realen Verhältnisse (die reale Welt) wiedergeben, d.h. die unterschiedlichen Objekte (Entitäten) und ihre Beziehungen. Ein **Datenmodell** ist dann das Muster (das Schema), nach dem die Daten logisch organisiert werden.
Im Hinblick zu den Anwenderprogrammen ist auf **Datenneutralität** und mit Blickrichtung auf die physische Datenbank auf **Datenunabhängigkeit** zu achten.

Datenneutralität bedeutet:
Neue Anwendungen und neue Benutzersichten sollen keinen Einfluß auf existierende Anwendungen und Sichten ausüben.
Datenunabhängigkeit bedeutet:
Neue Geräte, verbesserte Speichertechnologien, veränderte Zugriffspfade sollen sich in Anwendungen nur durch Leistungsverbesserung, nicht durch Funktionsänderung bemerkbar machen.

Bei vollständiger Datenneutralität ist es möglich, durch unterschiedliche Benutzersichten (Netze, Hierarchien, Relationen) das Basis-Schema zu betrachten. Eine Sicht (**view**) wird mit Hilfe der Datenmanipulationssprache (**DML**) aus der Basis, deren Struktur durch eine Datendefinitionssprache (**DDL**) festgelegt wurde, bestimmt.
Auf der anderen Seite sollte eine Reihe von Abbildungen das "Interne Schema" zur Beschreibung physischer Strukturen erzeugen. Sie enthalten mit ihren Katalogen die Details zu Zugriffspfaden und zur physischen Speicherung. Eine Speicherstrukturierungssprache (**SSL**) unterstützt die verschiedenen Alternativen zur physischen Speicherung.
Ein hohes Maß an Datenunabhängigkeit und -neutralität ist mit hierarchischen und auch mit netzwerkorientierten Datenbanken nicht zu erreichen.

Bsp.: Gegeben ist

RECHNUNG

RECH-NR.: 2010 DATUM: 31.12.1995
KUNDEN-NR.: 50118 NAME: ..... ADRESSE:
.....

ARTIKEL-NR.	BEZEICHNUNG	PREIS	MENGE	BETRAG
A	Widerstand	5,00	2000	10.000,00
B	Spule	3,00	1000	3.000,00
....	.....	.....	.....	
			SUMME:	18.000,00

Abb. 1.3-36: Ein Rechnungsformular

Das zugehörige Datenbankstrukturdiagramm einer netzwerkorientierten Datenbank, das die Daten des vorliegenden Rechunungsformulars benutzt, sieht so aus:

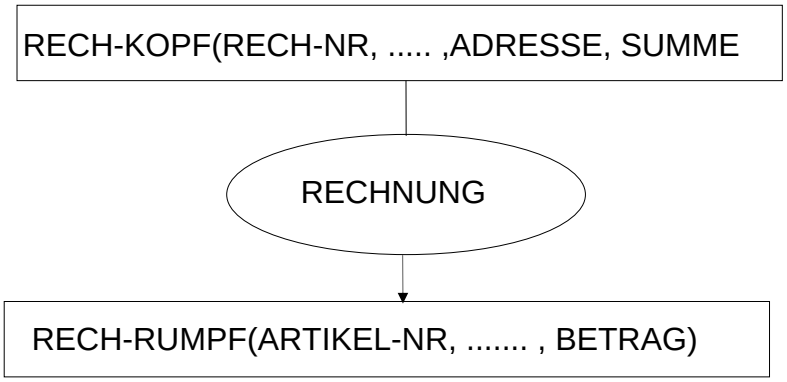


Abb. 1.3-37: Datenbankstrukturdiagramm zum Rechnungsformular

Besitzt dieser Entwurf für eine netzwerkorientierte Datenbank die Eigenschaften **Datenunabhängigkeit** und **Datenneutralität**?

Datenunabhängigeit?
Ein Zugriffspfad (, wesentlich bestimmt durch die Zeilenposition) ist zur Daten-wiedergewinnung unerläßlich.
Datenneutralität?
Es werden nur solche Anwendungen unterstützt, die zuerst den Rechnungskopf und dann den Rechnungsrumpf lesen.

Datenunabhängigkeit und Datenneutralität sind über ein netzwerkorientiertes DB-Modell nicht erreichbar[27].
Sind solche Eigenschaften mit dem relationalen DB-Modell möglich?
Da das Rechnungsformular die Struktur einer komplexen (geschachtelten) Tabelle aufweist, das relationale DB-Modell nur einfache Tabellen kennt, ist hier ein Nor-

[27] vgl.: Wedekind, Hartmut: "Relationale Datenbanksysteme", Informatik-Spektrum, Nr.5, !978, Seiten 5 - 16

malisierungsprozeß durchzuführen. Normalisieren bedeutet: Darstellung des logischen Schemas einer relationalen Datenbank in Form einfacher, geschachtelter Tabellen. Aus dem vorliegenden, hierarchischen Schema werden 2 Relationen:

RECH-KOPF(RECH-NR,KUNDEN-NR,DATUM,NAME,ADRESSE,SUMME)
RECH-RUMPF(RECH-NR,ARTIKEL-NR,BEZ,PREIS,MENGE,BETRAG)

Das Schema für die relationale Datenbank zeigt die gewünschte **Unabhängigkeit**: Eine Rumpfzeile kann angesprochen werden, ohne zum Kopf zugreifen zu müssen. Voraussetzung dafür ist: Es existiert ein Primärschlüssel im Rechnungskopf (RECH-NR).

1.3.6 Grundkonzepte von Datenmodellen

Ein Datenbankmodell (häufig auch nur als Datenmodell bezeichnet) ist ein System von Konzepten für Definition (und Evolution) von Datenbanken zur Festlegung von Syntax und Semantik von Datenbankschemata (Datenbeschreibungen) und Datenbankoperationen.

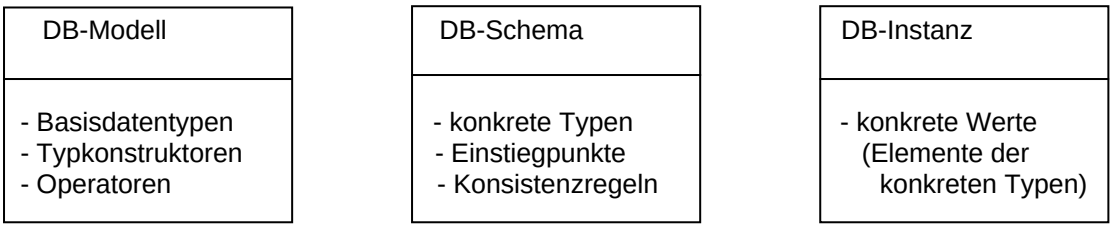


Abb.: Grundlegende Begriffe der Datenbankwelt

Das Datenbankmodell bietet neben Basisdatentypen eine Reihe von Typkonstruktoren zur Beschreibung des Datenbankschemas. Das Satenbankschema legt neben den in der Miniwelt vorkommenden (konkreten) Datentypen auch die Einstiegpunkte (im Relationenmodell die Namen der Tabellen) fest. Die mit diesen Datentypen assoziierten Operationen bestimmen, wie sich Ausprägungen der jeweiligen Datentypen, d. h. die konkreten Werte im Laufe des Datenbankbetriebs ändern können.
Im wesentlichen ist jedes konkrete Datenmodell charakterisiert durch:

Basisdatentypen
Definieren die elementaren Wertebereiche, aus denen komplexe Daten zusammengesetzt werden. Eine typischer Basisdatentyp wäre der Typ INTEGER für elementare Zahlenwerte.
Typkonstruktoren
Ermöglichen die Komposition aus elementaren (und anderen komplexen) Typen, z.B. der aus relationalen Datenbanken bekannte Typkonstruktor ROW zur Definition eines Tupeltyps. Ein Typkonstrutor erzeugt einen Datentyp
Typkompositionsregeln
Legen fest, in welcher Form sich Datentypen und Typkonstruktoren kombinieren lassen. Ein bekanntes Beispiel ist die Regel des relationalen Modells, die nur Typkonstruktoren in Form SET (ROW(Basisdatentypen)) zuläßt.

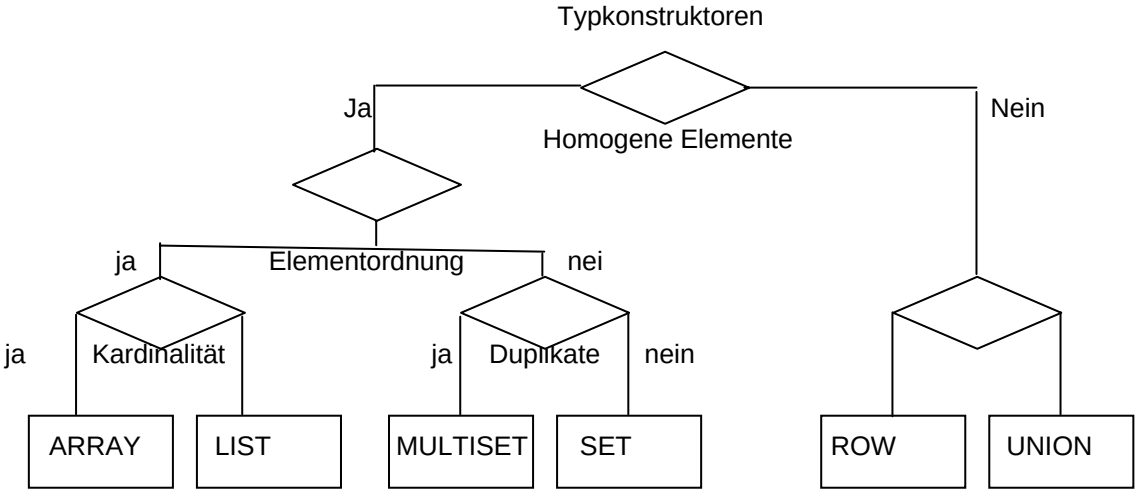


Abb. Überblick über (wertbasierte) Typkonstruktoren

1.3.7 Das Datenbankmodell für objektorientierte Datenbanken

1.3.7.1 Die Basis objektorientierter Datenbanknen

Grundlagen

Ein **objektorientiertes Datenbanksystem** ist auch nichts Anderes als ein Datenbanksystem mit den üblichen Leistungen (Datenintegration, Datenunabhängigkeit, Unterstützung vom Mehrbenutzerbetrieb, Gewährleistung der Datensicherheit, Datenschutz, Datenkonsistenz, Verarbeitungsintegrität). Allerdings stützt sich ein derartiges Datenbanksystem auf ein eigenes Datenbankmodell (objektorientiert, `OODM`) ab und darin liegt die Problematik: Eine einheitliche, allgemeingültige Definition für ein **OODM** gibt es noch nicht.
Ausgangspunkt ist das Ziel: Umweltsachverhalte beliebiger Art und Komplexität möglichst genau (`1:1` Abbildung) beim Datenbankentwurf wiederzugeben. Ein Umwelt-Objekt soll einem Datenbankobjekt entsprechen.
Auch komplex aufgebaute Objekte sollen modelliert werden können, denn Objekte können Bestandteile besitzen, die wiederum in Unterobjekten beschrieben werden können.
Im Schema werden, wie üblich Objekttypen spezifiziert. Zusätzlich zur Festlegung der gewöhnlichen Attribute (sog. einfache Datentypen: `integer`, `real`, `string`, `date`, `time`) und deren Wertebereiche ist hier noch anzugeben, von welchem Typ evtl. Unterobjekte sein können. Für den Aufbau komplexer Strukturen sollen außerdem nicht nur Konstruktoren für Tupel, sondern auch für Mengen[28] und Listen zur Verfügung stehen.
Zum Konzept für komplexe Objekte zählen auch entsprechende Operatoren. Sie dienen zum Auf- und Abbau (Konstruktoren und Destruktoren) und zum Zugriff auf die Eigenschaften von Objekten bzw. zur Bildung neuer Objekte aus existierenden Objekten (sog. Umsetzungsoperatoren). Objekte können nur über den Aufruf der definierten Operatoren (Methoden) angesprochen bzw. bearbeitet werden (**Konzept der Datenkapselung**). Lediglich über die Implementierung der

[28] z.B. ein Typkonstruktor zur Mengenbildung mit SET OF

Operatorprogramme können direkte Zugriffe auf die Wertepräsentation codiert werden.
Die Definition neuer Objekttypen schließt die für diese Typen charakterischen Operatoren (**Methoden**) mit ein. Objekktypen werden in diesem Zusammenhang **Klassen** genannt.
Objektorientierte Systeme unterstützen eine Reihe von Klassen (sog. Basisklassen), die häufig verwendete Datenstrukturen (Listen, Mengen, etc.) bereithalten. Basisklassen sind evtl. in Klassenbibliotheken verfügbar.
In einem OODM müssen alle Objekte eigenständig, von ihren aktuelle Werten unabhängig, identifiziert werden und zu diesem Zweck eine eindeutige, unveränderliche Kennung erhalten (**Objektidentifikator**, **Surrogat**). Dadurch ist der Anwender entlastet von der Festlegung eindeutiger, unveränderlicher Schlüssel für alle Objekte (, er darf dies auf Wunsch aber auch weiterhin tun). Jedes Objekt ist über seinen Identifikator ansprechbar. Gleichheit und Identität von Objekten können unterschieden werden.
Innerhalb des objektorientierten Datenbanksystems (OODBS) erfolgen alle Objektidentifizierungen über Surrogate. Das sind systemgenerierte, global eindeutige Bezeichner, die unabhängig vom physischen Speicherungsort sind (Ablageunabhängigkeit). Der Benutzer hat keinen Einfluß auf Surrogate. Benutzerschlüssel sind deshalb nicht überflüssig und können weiterhin zur Identifizierung von Objekten auf Benutzerebene interessant sein. Benutzerdefinierte Schlüssel zählen in objektorientierten Systemen zu den Eigenschaften des Objekts und haben nicht die Bedeutung wie in relationalen Systemen.
In einem OODM kann ein Typ Untertyp eines anderen sein. Der Untertyp ist eine **Spezialisierung** des Obertyps bzw. der Obertyp ist eine **Generalisierung** des Untertyps. Man spricht von der „IS_A“-Beziehung zwischen Unter- und Obertyp. Für den Untertyp müssen lediglich die gewünschten zusätzlichen Eigenschaften festgelegt sein, seine Objekte erben automatisch alle für den Obertyp definierten Eigenschaften.

Entwicklungslinien objektorientierter Datenbanksysteme

Man kann zwei Entwicklungslinien bei objektorientierten Datenbanksystemen (OODBS) [29] erkennen:

- Erweiterung bzw. Ergänzung objektorientierter Programmiersprachen (OOPL)
  Ausgangspunkt ist eine **OOPL** (z.B. C++, Smalltalk). Durch Ergänzung mit Datenbankkonzepten (Dauerhaftigkeit (Persistenz), Speicherstrukturen, Zugriffspfade, Transaktionen, Concurrency Control, Recovery-Mechanismen) wird ein OODBS entwickelt. Außerdem werden vordefinierte Klassen und Methoden mitgeliefert, die fehlenden Typkonstruktoren nachahmen (etwa SET OF) und Abfrageoperatoren bereitstellen (etwa das Auswählen von Objekten aus einer Menge nach bestimmten Bedingungen). Beispiele sind Datenbanken, die entweder auf Smalltalk[30]- oder C++[31]-Basis implementiert werden.
- Überlagerung relationaler Systeme mit Objektstrukturen
  Die bewährte relationale Datenbank-Technologie wird beibehalten und graduell um objektorientierte Eigenschaften ergänzt.

[29] vgl. Dittrich, Klaus R.: Objektorientiert, aktiv, erweiterbar: Stand und Tendenzen der "nachrelationalen" Datenbanktechnologie, it 5/90, Seiten 343 - 353
[30] Gemstone
[31] Object-Store, ONTOS, POET

Es gibt Neuentwicklungen voll objektorientierter Datenbankmodelle. Sie sind vollständig auf Erfordernisse der Datenbank und auf Objektorientierung zugeschnitten. Da genaue Kriterien für Konzepte objektorientierter Datenbanken erst seit 1989[32] vorliegen, sind Implementierungen noch nicht sehr zahlreich.

1.3.7.2 Grundlagen objektorientierter Systeme

Object Oriented Database Manifesto

In diesem Bericht[33] haben Datenbankforscher Richtlinien für das objektorientierte Datenbankmodell zusammengefaßt. Merkmale von objektorientierten Datenbanksysteme wurden hier in drei Kategorien eingeteilt: Pflchtmerkmale, optionale Merkmale, offene Merkmale.
Zu den aus der objektorientierten Programmierung entlehnten Pflichtmerkmalen zählen:

1. Objektbegriff / Objektidentität
In objektorientierten Datenbankmodellen besteht ein Objekt aus

- einem Objektidentifikator
- einer Menge von Werten (Attribute, Variable)
- einer Menge von Prozeduren / Operatoren[34]

Der Objektidentifikator wird vom DBMS vorgegeben[35] und werwaltet. Er ist systemweit eindeutig und während der Objektlebensdauer konstant.

2. Komplexe Objekte[36]
Ausgangspunkt sind elementare Werte, z.B. Zeichenketten oder Zahlen, aus denen mit Hilfe sog. Konstruktoren komplexe Werte gebildet werden. Einfache Konstruktoren sind:
- Tupelkonstruktor (Aggregation): Daten setzen sich häufig aus anderen Daten zusammen, auch die repräsentativen (physischen oder logischen) Gegenstände der relen Welt sind häufig aus anderen Gegenständen zusammengesetzt, z.B. das Aggregat `Adresse`:

[32] Atkison et al.: Object Oriented Database System Manifest
[33] Atkison, M.: The object oriented database manifesto, Proceedings First national Conference on Deductive and Object Oriented Databases, Kyoto 1989
[34] Die 3 Begriffe werden synonym verwendet
[35] Er beruht nicht auf den aktuellen Werten des Objekts (wie etwa der Primärschlüssel bei relationalen datenbanken)
[36] Synonym verwendete Begriffe: Strukturierte Objekte, Zusammengesetzte Objekte

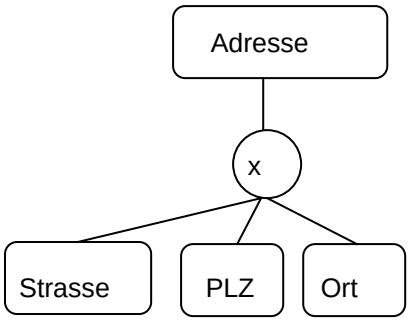


Abb.: Aggregat „Adresse“ in erweiterter ERM-Darstellungf

Häufig wird der Tupelkonstruktor mit „TUPLE OF“ notiert.

- Mengenkonstruktor (Gruppierung): Dadurch wird ein neuer Typ erzeugt, der aus mehreren Instanzen des in die Gruppierung eingehenden Typs eine Instanz des neuen Typs formt. Häufig wird der Mengenkonstruktor mit „SET OF“ notiert.

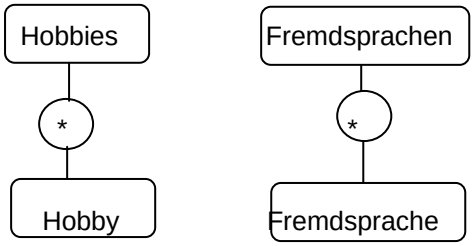


Abb.: „Gruppierungen in erweiterter ERM-Diagrammdarstellung

- Listenkonstruktor: „geordnete Menge“ (LIST OF)
-- erzeugt aus mehreren Elementen eines zugrundeliegenden Typs einen neuen Typ
-- Instanz des neuen Typs ist eine Liste aus Instanzen des zugrundeliegenden Typs (Elementtyp)
-- kann Elemente mehrfach beinhalten

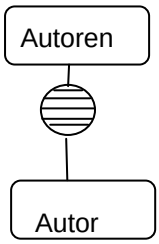


Abb.: „Listen in erweiterter ERM-Diagrammdarstellung

- Feldkonstruktor: „`array`“ ARRAY OF
-- ermöglicht indizierten Zugriff auf Elemente des zugrundeliegenden Elementtyps
-- immer feste Anzahl von Elementen (entgegen SET, LIST)
-- alle Elemente haben gleichen Elementtyp (entgegen Tupel)

Datenbankmodell für komplexe Werte: NF^2-Modell

3. <u>Klassen, Typen</u>
Eine Klasse ist eine Beschreibung, die
- einen Namen für die Objektmenge
- die Struktur der Objekte der Menge (ihre Werte)

- die Methoden, die die Objekte der Menge ausführen können
umfaßt. Ein Objekt einer Klasse wird Instanz genannt. Instanzen einer Klasse haben den gleichen Aufbau, benutzen diesselben Namen, Typen und Methoden. Aus der Klasse können Instanzen ins Leben gerufen werden.

4. Typhierarchie, Vererbung
Es können übergeordnete Klassen an untergeordnete Klassen Eigenschaften vererben. Die Eigenschaften können Zustand und / oder das Verhalten beschreiben.

5. Einkapselung
Zustand und Methoden sind für den Anwender gekapselt, d.h. für ihn unsichtbar (information hiding). Er kann nicht zu den Daten direkt zugreifen, sondern nur über die vorhandenen Methoden. Die Methoden sind ihm über Schnittstellen bekannt, über die er mit geeigneten Nachrichten die Methoden zum Arbeiten veranlassen kann.

6. Berechnungsvollständigkeit
Zur Realisierung der Methoden muß eine Sprache zur Verfügung stehen, die die Formulierung beliebiger Algorithmen gestattet. Die Datenbanksprache SQL in aktuell vorliegender Form genügt diesen Anforderungen nicht.

7. Überladen / Überschreiben und spätes Binden
Dabei handelt es sich um Detaikonzepte, die die Vorteile der Klassenhierarchie und Vererbung zur Geltung bringen.
Überladen wird durch die gleiche Namensvergabe für verschiedene Operatoren erzwungen.
Überschreiben ist eine Methode, die sowohl in der übergeordneten Klasse als auch in der untergeordneten Klasse definiert ist. Für die Instanzen der untergeordneten Klasse ist die Methode der untergeordneten Klasse maßgebend.
Spätes Binden bedeutet: Das Binden eines Operatornamens an den zugehörigen ausführbaren Programmcode erfolgt dynamisch erst zur Laufzeit.

Standard für objektorientierte Datenbanken: ODMG-93

Die Mitgliederfirmen der ODMG (Object Database Management Group)[37] haben einen Standard für objektorientierte Datenbanksysteme (ODBS)[38] definiert mit Standardisierungsvorschlägen für die Bereiche

- Objektmodell
- Objekt-Definitionssprache (**ODL**)
- Objekt-Abfragesprache (**OQL**)
- C++- und Smalltalk-Anbindung für **ODL** und **OQL**, sowie **OML** (Object Manipulation Language)

Anbindung für Programmiersprachen

Eine ODMG-ODBS sieht dafür folgende Möglichkeiten vor

[37] eine Gruppe führender Hersteller objektorientierter Datenbanksysteme
[38] Spezifikation ODMG-93

- Definition von Datenbankschemata
- Einfüllen von Daten
- Zugriff und Manipulation von Daten
- Verwendung der OQL

Das „Typ“-System der Programmiersprache und das Datenbankmodell des Datenbanksystems sollen möglichst integriert werden können. So soll das DBS mit dem erweiterten „Typ“-System der jeweiligen Programmiersprache definiert werden können.

1.3.7.3 Objektrelationale Datenbanken

Es handelt sich dabei im Kern um relationale Datenbanken, die um bestimmte objektorientierte Konzepte erweitert werden. Dazu gehören Objektidentität. Typhierarchien und Typkonstruktoren. Das ermöglicht die Konstruktion benutzerdefinierter Datentypen, z.B.:

```
create type adresse_t as object (
 strasse varchar2(20),
 hausnummer number(3),
 ortsname varchar2(30));
/
```

1.3.7.3.1 Objektrelationale Konzepe

Zu den wesentliche objektrelationalen Konzepten zählen:

- Typkonstruktoren
- benutzerdefinierte Datentypen
- Typhierarchien
- Methoden
- Objektidentifikatoren
- benutzerdefinierte Ordnungen
- typisierte Tabellen bzw. Objekttabellen
- Tabellenhierarchien
- typisierte Sichten bzw. Objektsichten
- Sichthierarchien

Objektrelationales Typsystem

Das Typsystem eines objektrelationalen DBMS bietet neben einer Reihe von Basisdatentypen eine Menge von Typkonstruktoren zur Erzeugung komplexer Datentypen an.

1. Basisdatentypen
Basisdatentypen werden für folgende Arten von Wertebereichen bereitgestellt:

Wahrheitswerte, ganze Zahlen, Festkommazahlen, Fließkommazahlen, alphanumerische Zeichenketten, binäre Zeichenketten, Datum und Zeitangaben

Vielfach zählen auch LOB[39]-Datentypen zu den objektrelationalen Konzepten. LOB-Datentypen ermöglichen die Verwaltung beliebig langer binärer und

[39] LOB steht für Large Object

alphanumerischer Zeichenketten, wie sie in Multimedia-Anwendungen gebraucht werden.

2. Typkonstruktoren
Typkonstruktoren ermöglichen die Konstruktion neuer Datentypen. Durch geschachtelte Anwendung von Typkonstruktoren können beliebig komplexe Datentypen entstehen. Zu den allgemein bekannten Typkonstruktoren gehören:

ROW	Tupeltypkonstruktor
ARRAY	Arraytykonstruktor
MULTISET	Multimengentypkonstruktor[40]
SET	Listentypkonstruktor
OBJECT	Objecttypkonstruktor
REF	Referenztypkonstruktor

3. Benutzerdefinierte Datentypen (User Defined Types, UDTs)
In aktuellen relationalen Datenbanken werden 2 Arten von benutzerdefinierten Datentypen unerschieden: Distinct Typen und Strukturdatentypen.
Distinct-Typen (User Defined Distinct Types) ermöglichen die Definition von Datentypen, die Kopien von bereits existierenden Datentypen darstellen. Wegen der strengen Typisierung sind Instanzen eines Distinct Typs nicht mit den Instanzen des zugrundeliegenden Quelltyps bzw. mit Instanzen eines anderen Distinct Typs vergleichbar. Bspw. ist ein Wert vom Typ Euro nicht mit einem Typ Franken vergleichbar, auch wenn beide Datentypen eine Kopie desselben (numerischen) Datentyps darstellen. Ist eine Umwandlung von einem Datentyp zu einem anderen Datentyp gewünscht, so muß eine geeignete Cast-Funktion für die Umwandlung implementiert werden.
Folgende Strukturdatentypen (User Defined Structured Types) sind in objektrelationalen Datenbanksystemen bekannt:

- benannte Tupeltypen (Informix)
  definieren eine Struktur, die aus Tupelfeldern besteht. Die Instanzen von benannten Tupeltypen besitzen keine Methoden und werden nicht mit OIDs assoziiert.
- strukturierte Typen (DB2 und Standard-SQL)
  ermöglichen die Definition von Datentypen, die eine Menge von Attributen und Methoden umfassen, besitzen allerdings keine inhärente OID.
- Objekttypen (Oracle)
  definieren Datentypen, die eine Menge von Attributen und Methoden umfassen. Jedes Objekt besitzt eine inhärente, systemvergebene, unveränderliche OID, über die systemweit eindeutig identifiziert und registriert werden kann.
  Das in Oracle umgesetzte Konzept der Objekttypen, sieht allerdings OIDs nur für Objekte vor, die als Zeilen einer Objekttabelle gespeichert werden.

Strukturdatentypen können wie andere Datentypen, auch als Typ eines Attributs bzw. eines Tupelfelds dienen. Auf diese Weise können Objektansammlungen, die eine Komposition in UML entprechen, direkt in die Datenbank umgesetzt werden. Eng verbunden mit dem Konzept der Strukturdatentypen sind folgende Konzepte: Typhierarchien, Methoden, Objektidentifikationen

4. Typhierarchien
Eine Typhierarchie entsteht durch Definition von Subtypen. In aktuellen objektrelationalen Datenbanken kann ein Subtyp nur von einem Strukturdatentyp

[40] in Oracle heißt der Multimengentypkonstruktor TABLE statt MULTISET

abgeleitet werden. Folglich kommen nur folgende Arten von Datentypen als Supertypen und damit auch als Subtypen in Frage:

- benannte Tupeltypen (Informix)
- strukturierte Typen (DB2 und Standard SQL)
- Objekttypen (Oracle)

Ein Subtyp erbt jeweils die Struktur (Attribute) und das Verhalten (Methoden) seines Supertyps).

5. Methoden
Methoden implementieren das Verhalten des jeweiligen Strukturdatentyps. Prinzipiell ermöglichen sie das Speichern und Ausführen von Programmcode in DBMS. Generell werden 3 Arten von Methoden unterschieden : Instanzmethoden, statische Methoden, Konstruktormethoden.

Objektrelationale Tabellen

Das Konzept einer Tabelle stellt auch im objektrelationalen Datenbanksystem die einzige persistente Datenstruktur dar. Neben normalen Tabellen gibt es typisierte Tabellen, die auf einem Strukturdatentyp basieren.

1. Tupeltabellen
Eine Tupeltabelle beschreibt eine persistente Multimenge von Tupeln. Der Typ einer Spalte in einer Tupeltabelle ist mit der objektrelationalen Erweiterung des Typsystems nicht mehr nur auf Basistypen beschränkt. Es kann ein beliebiger Datentyp des Typsystems sein.

2. Typisierte Tabellen
Eine typisierte Tabelle bezeichnet eine Tabelle, deren Typ mit Hilfe eines Strukturdatentyps festgelegt wird.
Typisierte Tabellen in Oracle: Eine typisierte Tabelle stützt sich in Oracle auf einen Objekttyp ab. Oracle versteht daher unter typisierten Tabellen Objekttabellen.

1.3.7.3.2 Objektrelationale Unterstützung in SQL3

Die SQL/Object-Spezifikation erweitert SQL-92 um objektrelationale Fähigkeiten. Neue Datentypen sind boolsche Zeichen und binäre große Objekte (**LOB**s).
Objekte in SQL3. Im Rahmen der SQL-Foundation- und Object-Spezifikation erlaubt SQL benutzerdefinierte Datentypen, Typkonstruktoren, Kollektionstypen, benutzerdefinierte Funktionen und Prozeduren, Unterstützung großer Objekte und Trigger. In SQL3 gibt es zwei Objekttypen:

- Row- oder Tupel-Typen, deren Instanzen Tupel in Tabellen sind
- Abstrakte Datentypen (ADT), d.h. generelle Typen, die als Komponenten von Tupeln benutzt werden.

1.3.8 Das UML-Modell

Die Unified Modelling Language (UML)[41] ist eine objektorientierte Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen von Softwaresystemen. Sie besteht aus verschiedenen Diagrammen, die wiederum verschiedene grafische Elemente besitzen. Die sprachunabhängige, grafische Notation unterstützt den gesamten Softwareprozeß, u.a. die folgenden Aufgaben:

- Analyse der Anforderungen
- Visualisierung eines Problems
- Kommunikation mit Anwendungsexperten
- Implementierung zur Lösung in einer Programmiersprache
- Vorbereitung der Dokumentation

Die Bedeutung (, also die Semantik,) der Elemente ist genau festgelegt. Innerhalb der UML gibt es allerdings für ein und denselben Sachverhalt mehrere Darstellungsarten. Zur Modellierung eines Systems dient das Objektmodell, das im wesentlichen den gängigen Prinzipien der Objektorientierung folgt: Objekte gleicher Art werden zu Klassen zusammengefasst, die den Typ dieser Objekte definieren. Assoziationen erlauben die Angabe von Beziehungen zwischen Objekten bzw. Klassen.
Für den Datenbankentwurf ist die strukturierte Modellierung des Systems bestehend aus Objektklassen und Assoziationen zwischen den Klassen am wichtigsten. Objekte entsprechen den Entities und **Objektklassen** beschreiben eine Menge von gleichartigen Objekten / Entities. Zusammenhänge (Beziehungen) zischen Entities werden als **Assoziationen** zwischen Klassen beschrieben.

1.3.8.1 Die Diagramme

Die Notation der UML umfaßt Diagramme für die Darstellung verschiedener Ansichten auf das System. Sie sind vergleichbar mit Bauplänen.

Diagramm	Einsatzgebiet
Use-Case	Geschäftsprozesse, allgemeine Einsatzmöglichkeiten
Klassendiagramm	So gut wie überall, das Klassendiagramm ist das wichtigste Diagramm der UML
Interaktionsdiagramm	Zeigt den Nachrichtenfluß und damit die Zusammenarbeit der Objekte im zeitlichen Ablauf
- Sequenzdiagramm	Zeitliche Aufrufstruktur mit wenigen Klasse
- Kolloborationsdiagramm	Zeitliche Aufrufstruktur mit wenigen Nachrichten
Package-Diagramm	Groborientierung, in welchem Modul welche Klasse zu finden ist. Aufteilung in Unterprojekte, Bibliotheken, Übersetzungseinheiten.
Zustandsdiagramm	Darstellung des dynamischen Verhaltens
Aktivitätsdiagramm	Bei parallelen Prozessen und anderer Parallelität, Geschäftsprozesse
Implementierungsdiagramm	Besonders für die Darstellung von verteilten Anwendungen und Komponenten; allgemein Darstellung von Implementierungs-einheiten (Übersetzungseinheiten, ausführbare Programme, Hardwarestruktur)

[41] 1996 begannen die Arbeiten an der UML. Im Jahre 1997 wurde die Version 1.0 bei der Object Management Group (OMG) als Standardisierungsvorschlag eingereicht, und eine Beschreibung der Version 1.0 wurde veröffentlicht. Im September 1991 wurde die Version 1.1. bei der OMG eingereicht.

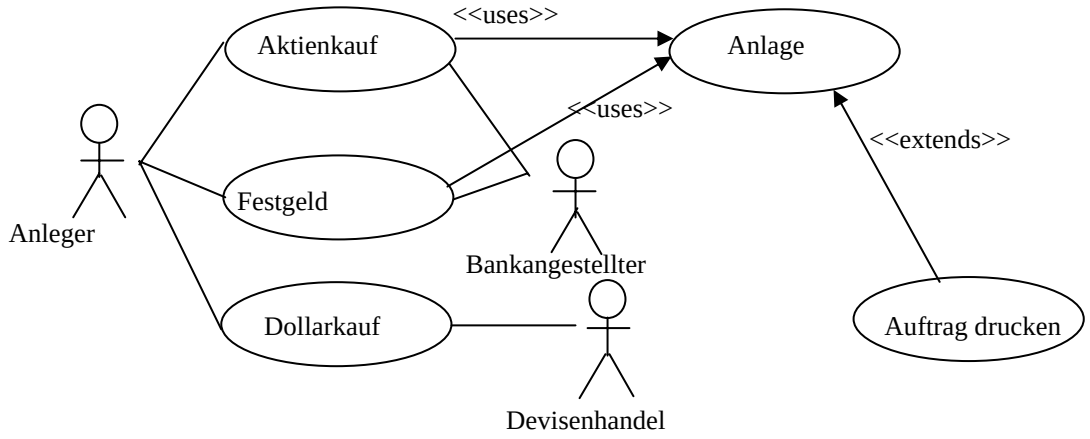
- Komponentendiagramm	Zusammenhänge der Software
- Deployment-Diagramm	Hardwareaufbau

Abb.: Einsatzgebiete und Eigenschaften der verschiedenen UML-Diagramme

Jedes der Diagramme besitzt zahlreiche Stilelemente mit verschiedenen Beschriftungsarten[42]. Das wesentliche Merkmal der UML ist die Integration der verschiedenen Diagrammarten, die sowohl die statischen als auch die dynamischen Komponenten des zu entwerfenden Systems beschreiben.
Neben Objekt- und Klassendiagrammen, die für die Modellierung der statischen Komponenten eingesetzt werden können, gibt es die Use-Case-, Aktivitäts-, Datenfluss- und Sequenzdiagramme zur Spezifikation der dynamischen Komponenten.

Use Case

UML bietet für die dem Software-Entwurf vorgelagerte Phase der Anforderungsanalyse eine grafische Modellierungsmethode zur Definition von Anwendungsfällen (typische Anwendungssituationen). Anwendungsfälle kann man sich als eine Sammlung von Szenarios vorstellen. Jedes Szenario beschreibt eine Sequenz von Schritten. Jede dieser Sequenzen wird von einem Menschen, einem anderen System, einem Teil der Hardware oder durch Zeitablauf in Gang gesetzt.
Entitäten, die solche Sequenzen anstoßen, nennt man Akteure. Das Ergebnis dieser Sequenz muß etwas sein, was entweder dem Akteur, der sie iniziierte oder einem anderen Akteur nutzt.
Ein „Use Case“ ist eine typische Handlung, die ein Benutzer mit dem System ausführt, z.B. Aktienkauf. Im Diagramm werden Use Cases durch Ellipsen dargestellt. Verbindungen zu den entsprechenden Use-Cases werden durch Linien dargestellt. Damit wird angezeigt, welche Aktoren an dem entsprechenden Use Case beteiligt sind.
Bsp.: „Geld anlegen“



In diesem Use-Case-Diagramm gibt es zwei weitere Verbindungstypen: <<uses>> und <<extends>>.
<<uses>> wird verwendet, wenn zwei oder mehrere Use-Cases einen ähnlichen Aufbau haben und Verhalten durch Kopieren wiederholt dargestellt werden muß. Mit <<uses>> ist es möglich, Verhalten von den Use-Cases in einen seperaten Use-Case zu verlagern.
Bei <<extends>> wird das Verhalten erweitert. So stellt „Auftrag drucken“ dem Use-Case Anlage zusätzliche Funktionalität zur Verfügung

[42] Vgl.: Günther Wahl: UML kompakt, OBJEKTspektrum 2/1998

Abb.: Use-Case „Geld anlegen“

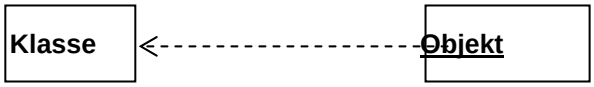
<<uses>> bzw. <<extends>> sind Darstellungen spezieller Stereotypen. Stereotypen sind Erweiterungsmechanismen der UML. Sie haben etwa die Bedeutung „so ähnlich etwa wie“und erlauben eine weitere Einteilung der Klassen in Schnittstellen-, Kontroll- und Entitätenobjekte (irgendwelche Dinge).

Elemente und Darstellung des Klassendiagramms

Das Klassendiagramm beschreibt die statische Struktur der Objekte in einem System sowie ihre Beziehungen untereinander. Die **Klasse** ist das zentrale Element. Klassen werden durch Rechtecke dargestellt, die entweder den Namen der Klasse tragen oder zusätzlich auch Attribute und Operationen. Klassenname, Attribute, Operationen (Methoden) sind jeweils durch eine horizontale Linie getrennt. Klassennamen beginnen mit Großbuchstaben und sind Substantive im Singular.
Ein strenge visuelle Unterscheidung zwischen Klassen und Objekten entfällt in der UML. Objekte werden von den Klassen dadurch unterschieden, daß ihre Bezeichnung unterstrichen ist. Auch können Klassen und Objekte zusammen im Klassendiagramm auftreten.



Wenn man die Objekt-Klassen-Beziehung (Exemplarbeziehung, Instanzbeziehung) darstellen möchte, wird zwischen einem Objekt und seiner Klasse ein gestrichelter Pfeil in Richtung Klasse gezeichnet:



Die Definition einer Klasse umfaßt die „bedeutsamen“ Eigenschaften. Das sind:

- Attribute
  d.h.: die Struktur der Objekte: ihre Bestandteile und die in ihnen enthaltenen Informationen und Daten.. Abhängig von der Detaillierung im Diagramm kann die Notation für ein Attribut den Attributnamen, den Typ und den voreingestellten Wert zeigen:
  Sichtbarkeit Name: Typ = voreingestellter Wert
- Operationen
  d.h.: das Verhalten der Objekte. Manchmal wird auch von Services oder Methoden gesprochen. Das Verhalten eines Objekts wird beschrieben durch die möglichen Nachrichten, die es verstehen kann. Zu jeder Nachricht benötigt das Objekt entsprechende Operationen. Die UML-Syntax für Operationen ist:

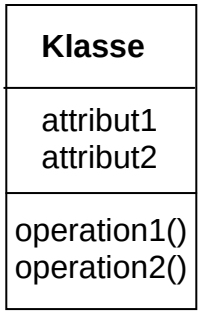
  Sichtbarkeit Name (Parameterliste) : Rückgabetypausdruck (Eigenschaften)

  Sichtbarkeit ist + (öffentlich), # (geschützt) oder – (privat)
  Name ist eine Zeichenkette
  Parameterliste enthält optional Argumente, deren Syntax dieselbe wie für Attribute ist
  Rückgabetypausdruck ist eine optionale, sprachabhängige Spezifikation
  Eigenschaften zeigt Eigenschaftswerte (über String) an, die für die Operation Anwendung finden
- Zusicherungen
  Die Bedingungen, Voraussetzungen und Regeln, die die Objekte erfüllen müssen, werden Zusicherungen genannt. UML definiert keine strikte Syntax für die Beschreibung von Bedingungen. Sie müssen nur in geschweifte Klammern ({}) gesetzt werden.

Idealerweise sollten regeln als Zusicherungen (engl. assertions) in der Programmiersprache implementiert werden können.
Bsp.: Eigenschaften eines Kreises
Zu den Eigenschaften eines Kreises, der auf einem Bildschirm dargestellt werden soll, gehören:

- Attribute: Radius des Kreises, Position des Kreises auf dem Bildschirm
- Operationen: Anzeigen und Entfernen des Kreises, Verschieben des Kreises, Verändern des Radius des Kreises
- Zusicherungen: Der Radius darf nicht negativ sein und nicht Null sein (radius>0)

Attribute werden mindestens mit ihrem Namen aufgeführt und können zusätzliche Angaben zu ihrem Typ (d.h. ihrer Klasse), einen Initialwert und evtl. Eigenschaftswerte und Zusicherungen enthalten. Attribute bilden den Datenbestand einer Klasse.
Operationen (Methoden) werden mindestens mit ihrem Namen, zusätzlich durch ihre möglichen Parameter, deren Klasse und Initialwerte sowie evtl. Eigenschaftswerte und Zusicherungen notiert. Methoden sind die aus anderen Sprachen bekannten Funktionen.



Bsp.: „Automatische Geldausgabe“

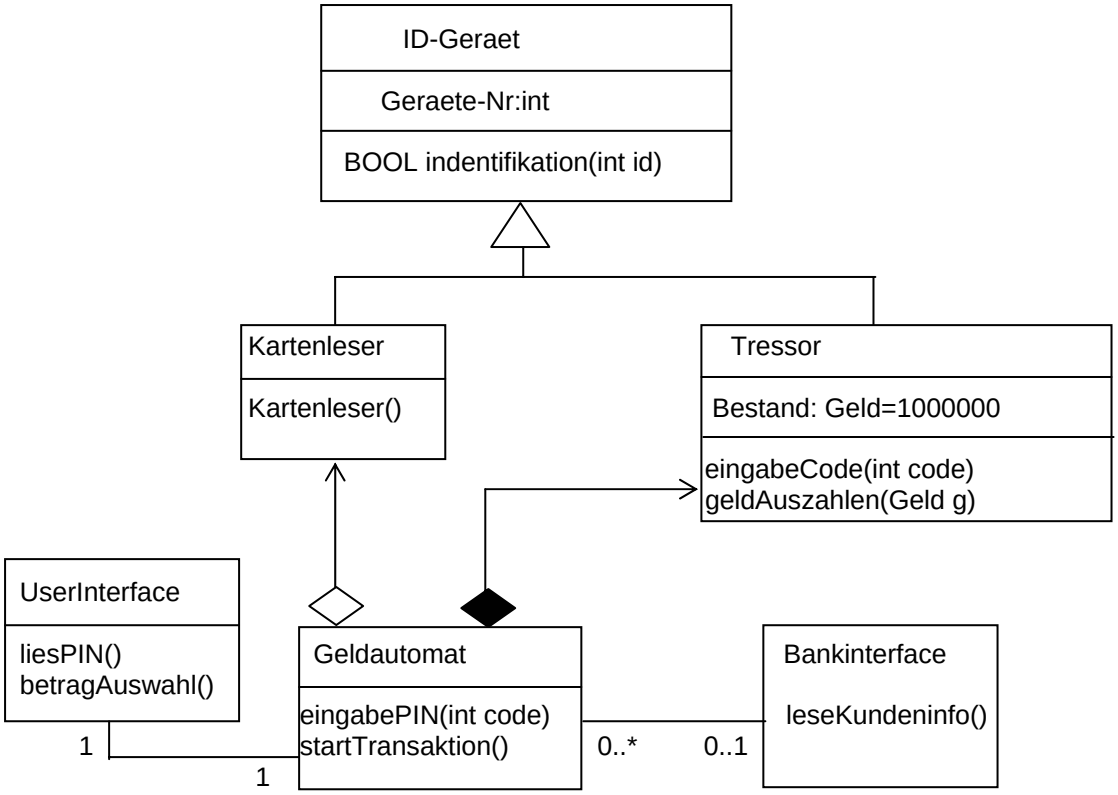
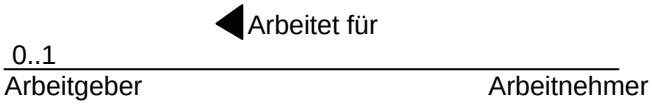


Abb.: Klassendiagramm „automatische Geldausgabe“

Die Klasse ist das zentrale Element, sie wird als Rechteck dargestellt (z.B. Geldautomat“). Die gefundenen Klassen sind durch Linien miteinander verbunden. Diese Linien stellen die Elemente Assoziation, Aggregation, Komposition und Vererbung dar.

Eine Assoziation (association) ist eine Strukturbeziehung, die eine Menge von Objektbeziehungen beschreibt. Eine Aggregation ist eine Sonderform der Assoziation. Sie repräsentiert eine (strukturelle) Ganzes/Teil-Beziehung. Grafisch wird eine Assoziation als durchgezogene Line wiedergegeben, die gerichtet sein kann, manchmal eine Beschriftung besitzt und oft noch weitere Details wie z.B. Muliplizität (Kardinalität) oder Rollenanmen enthält, z.B.:



Eine Assoziation kann einen Namen zur Beschreibung der Natur der Beziehung („Arbeitet für“) besitzen. Damit die Bedeutung unzweideutig ist, kann man dem Namen eine Richtung zuweisen: Ein Dreieck zeigt in die Richtung, in der der Name gelesen werden soll.
Rollen („Arbeitgeber, Arbeitnehmer) sind Namen für Klassen in einer Relation. Eine Rolle ist die Seite, die die Klasse an einem Ende der Assoziation der Klasse am anderen Ende der Assoziation zukehrt. Die Navigierbarkeit kann durch einen Pfeil in Richtung einer Rolle angezeigt werden.

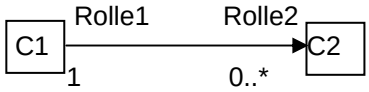


Abb.: Binäre Relation R = C1 x C2

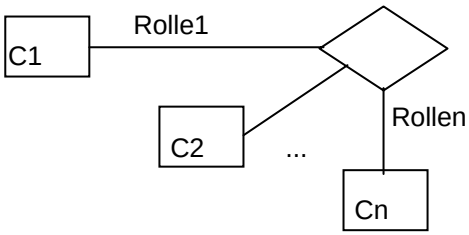


Abb.: n-äre Relation C1 x C2 x ... x Cn

In vielen Situationen ist es wichtig anzugeben, wie viele Objekte in der Instanz einer Assoziation miteinander zusammenhänen können. Die Frage „Wie viele?“ bezeichnet man als Multiplizität der Rolle einer Assoziation. Gibt man an einem Ende der Assoziation eine Multiplizität an, dann spezifiziert man dadurch: Für jedes Objekt am entgegengesetzten Ende der Assoziation muß die angegebene Anzahl von Objekten vorhanden sein.

1:1	1..*
1:1..n	0..* 2..6
0..n:2..6	0..* *
0..n:0..n	17 4
17:4	n m
?	

Abb.: Kardinalitäten für Beziehungen

Jede Assoziation[43] kann eine Richtung besitzen. Diese bestimmt ein Pfeil am Ende der Assoziation. Zugriffe können dann nur in Pfeilrichtung (Navigationsfähigkeit) erfolgen.
Reflexive Assoziationen. Manchmal ist eine Klasse auch mit sich selbst assoziiert. Das kann der Fall sein, wenn die Klasse Objekte hat, die mehrere Rollen spielen können. Ein Fahrzeuginsasse kann entweder Fahrer oder Beifahrer sein. In der Rolle des Fahrers fährt bspw. ein Fahrzeuginsasse „null“ oder „mehrere Fahrzeuginsassen“, die die Rolle von Beifahrern spielen.

[43] Eine Navigation mit Pfeil kann als Zeiger in einer Programmiersprache betrachtet werden.

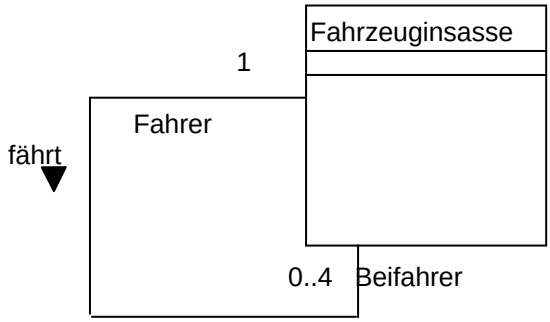


Abb.:

Bei einer refleviven Assoziation zieht man eine Linie von der Klasse aus zu dieser zurück. Man kann die Rollen sowie die Namen, die Richtung und die Multiplizität der Assoziation angeben.
Abhängigkeiten. Manchmal nutzt eine Klasse eine andere. Das nennt man Abhängigkeit (dependency). Die UML-Notation ist eine gestrichelte Linie mit einem Pfeil, z.B.:.

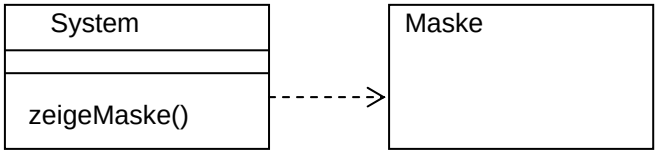


Abb.:

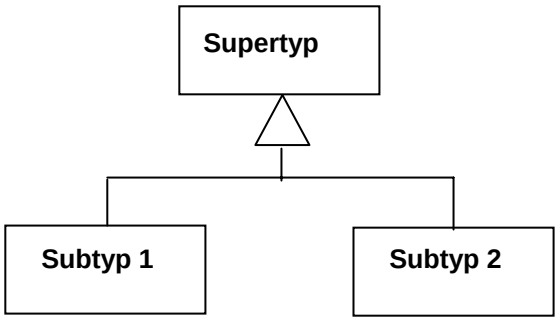
Eine Aggregation wird durch eine Raute dargestellt, z.B. zwischen „Kartenleser“ und „Geldautomat“[44]. Sie gibt an:

Die Klasse „Kartenleser“ ist in der Klasse „Geldautomat“ enthalten (**Ist-Teil-von-Beziehung**)

Die Komposition wird durch eine ausgefüllte Raute dargestellt und beschreibt ein „physikalisches Enthaltensein“. Die Kompositionsbeziehung ist eine strengere Form der Aggregationsbeziehung, bei der die Teile vom Aggregat existenzabhängig sind und ein Teil höchstens einem Aggregat zugeordnet werden darf. Ein Teil existiert höchstens so lange wie sein Aggregat, es sei denn, es wird vor dem Vernichten des Aggregats an ein anderes Aggregat weitergereicht. Auf der Seite der Aggregat-Klasse kann eine der beiden Kardinalitäten 0..1 oder 1 angegeben werden. Im Zusammenhang mit einer Kompositionsbeziehung drückt die Kardinalität 0..1 aus, dass ein Teil so lange existenzunabhängig ist, bis es einem Aggregat zugeordnet wird. Danach muß es wie bei der Kardinalität 1 immer einem Aggregat zugeordnet sein.

Die **Vererbung** (Spezialisierung bzw. Generalisierung) stellt eine Verallgemeinerung von Eigenschaften dar. Eine Generalisierung (generalization) ist eine Beziehung zwischen dem Allgemeinen und dem Speziellen, in der Objekte des speziellen Typs (der Subklasse) durch Elemente des allgemeinen Typs (der Oberklassse) ersetzt werden können. Grafisch wird eine Generalisierung als durchgezogene Linle mit einer unausgefüllten, auf die Oberklasse zeigenden Pfeilspitze wiedergegeben, z.B.:

[44] vgl. Abb.: Klassendiagramm „automatische Geldausgabe“.



Schnittstellen und abstrakte Klassen. Eine Schnittstelle ist eine Ansammlung von Operationen, die eine Klasse ausführt.. Eine Klasse hängt mit der Schnittstelle über die Implementierung zusammen. Eine Schnittstelle modelliert man in der Form einer gestrichelten Linie mit einem großen, unausgefüllten Dreieck, das neben der Schnittstelle steht und auf diese zeigt. Bei der Bildung einer Unterklasse wird beides vererbt. Eine reine Schnittstelle (wie bspw. in Java) ist eine Klasse ohne Implementierung und besitzt daher nur Operationsdeklarationen. Schnittstellen werden oft mit Hilfe abstrakter Klassen deklariert.
Bei abstrakten Klassen oder Methoden wird der Name des abstrakten Gegenstands in der UML u.U. kursiv geschrieben. Ebenso kann man die Bedingung {abstract} benutzen.

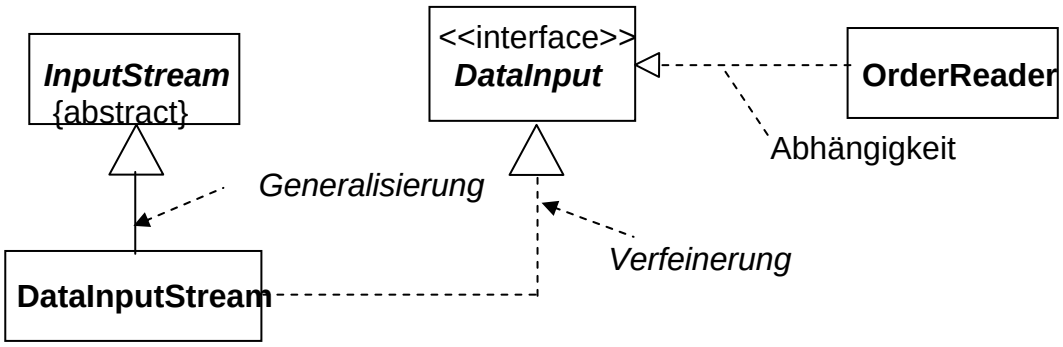


Abb.: Schnittstellen und abstrakte Klassen: Ein Beispiel aus Java

Irgendeine Klasse, z.B. „OrderReader“ benötigt die DataInput-Funktionalität. Die Klasse DataInputStream implementiert DataInput und InputStream. Die Verbindung zwischen DataInputStream und DataInput ist eine „Verfeinerung (refinement)“. Eine Verfeinerung ist in UML ein allgemeiner Begriff zur Anzeige eines höheren Detaillierungsniveaus.
Die Objektbeziehung zwischen OrderReader und DataInput ist eine Abhängigkeit. Sie zeigt, daß „OrderReader“ die Schnittstelle „DataInput für einige Zwecke benutzt.
Abstrakte Klassen und Schnittstellen ermöglichen die Definition einer Schnittstelle und das Verschieben ihrer Implementierung auf später. Jedoch kann die abstrakte Klasse schon die Implementierung einiger Methoden enthalten, während die Schnittstelle die Verschiebung der Definition aller Methoden erzwingt.
Eine andere Darstellung einer Klasse und einer Schnittstelle besteht aus einem (kleinen) Kreis, der mit der Klasse durch eine Linie verbunden ist, z.B.:

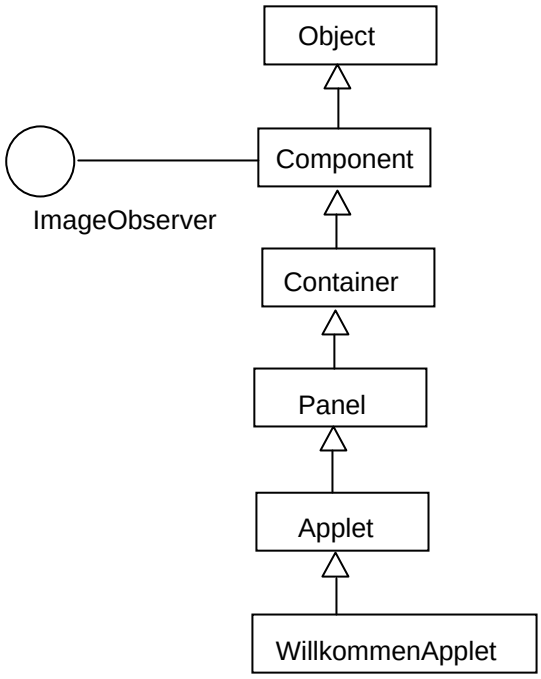


Abb.: Vererbungshierarchie von WillkommenApplet einschl. Schnittstelle ImageObserver

Interaktionsdiagramm

Es gibt zwei Arten von Interaktionsdiagrammen:

- Sequenzdiagramme
- Kollaborationsdiagramme[45]

Die beiden Diagramme beschreiben zeitliche Abläufe, d.h. Aufrufsequenzen. Ausgangspunkt von Interaktionsdiagrammen sind Fallbeispiele (Szenarios). Beim Erstellen der Diagramme konzentriert man sich auf die wichtigsten Fälle. Anschließend werden die Sonderfälle mit einbezogen.

1. Sequenzdiagramme

Ein Sequenzdiagramm ist ein Interaktionsdiagramm, bei dem die zeitliche Abfolge der Nachrichten im Vordergrund steht. Sequenzdiagramme visualisieren

- die Lebensdauer von Objekten (gestrichelte Linie, die die Existenz eines Objekts während eines Zeitraums darstellt). Die Zeitachse verläuft von oben nach unten. Objekte sind als Rechtecke am oberen Rand von gestrichelten (Lebens-) Linien dargestellt. Am linken Rand können noch Kommentare stehen.
- die Aktivität von Objekten. Der Focus-of-Control (das langezogene Rechteck) gibt an, wo eine Aktivität ausgeführt wird. Ist ein Objekt ohne Aktivität vorhanden, wird dies durch eine gestrichelte Linie angezeigt.Objekte werden durch einen Pfeil auf das Rechteck erzeugt, ein Löschen wird durch ein Kreuz dargestellt.
- Nachrichenaustauch zwischen Objekten (beschriftete Pfeile). Über den Pfeilen stehen Operationen, die ausgeführt werden. In eckigen Klammern können Bedingungen angegeben werden, wann die Operation ausgeführt wird. Ruft das Objekt eine Operation von sich selbst auf, dann zeigt der Pfeil auf die Lebenslinie zurück.

[45] Kollaboration: Die Gesamtheit der Objekte, die bei der Ausführung einer Aufgabe interagieren, einschl. der Verknüpfung zwischen ihnen.

Die in den Anwendungsfällen beschriebenen Objektinteraktionen können in Sequenzdiagrammen verfeinert werden. Für jeden Anwendungsfall können ein oder mehrere Sequenzdiagramm(e) konstruiert werden. Sequenzdiagramme stammen aus der Telekommunikationstechnik (z.B. für Protokolle).
Bsp.: Ablauf des „Use Cases“: „Neue Vorlesung anbieten“

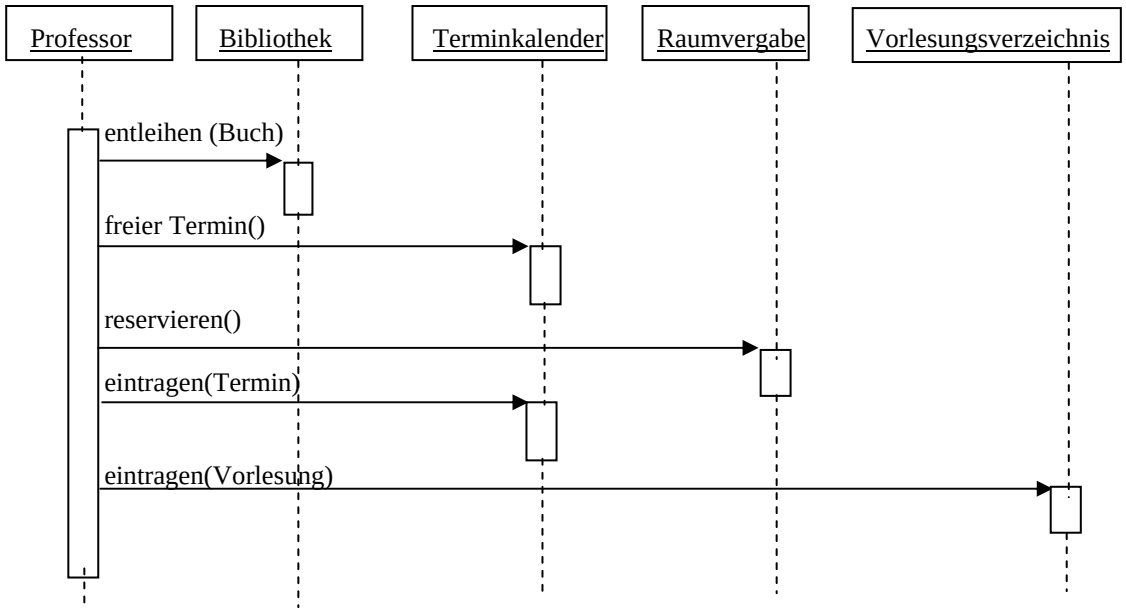


Abb.: Sequenzdiagramm: „Neue Vorlesung anbieten“

2. Kollaborationsdiagramme

Ein Kollaborationsdiagramm ist ein Interaktionsdiagramm, in dem die strukturelle Organisation der Objekte im Vordergrund steht, die Nachrichten senden bzw. empfangen.
Sequenz- und Kollaborationsdiagramme sind isomorph, so daß man das eine in das andere umwandeln kann. Sequenzdiagramme haben zwei Merkmale, die sie von Kolloborationsdiagrammen unterscheiden:

1. die Objektlebenslinie (senkrechte gestrichelte Linie)
2. Kontrollfokus (langes, schmales Rechteck)

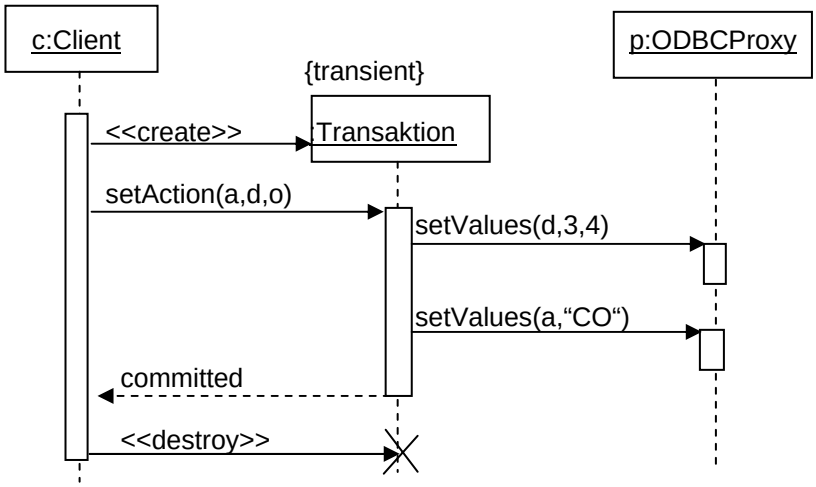


Abb.: Sequenzdiagramm

Kollaborationsdiagramme haben zwei Merkmale, die sie von Sequenzdiagrammen unterscheiden:

1. Zur Anzeige, wie ein Objekt mit einem anderen Objekt verbunden ist, kann man einen Pfad Stereotyp am entferntesten Ende der Objektbeziehung angeben (wie <<local>> zur Kennzeichnung, dass dieses Objekt für das zu sendende Objekt lokal ist).
2. Zur Kennzeichnung der zeitlichen Anordnung schreibt man eine Nummer als Präfix vor die Nachricht (beginnend mit der Nummer 1, monoton wachsend für jede weitere Nachricht für den Kontrollfluss).

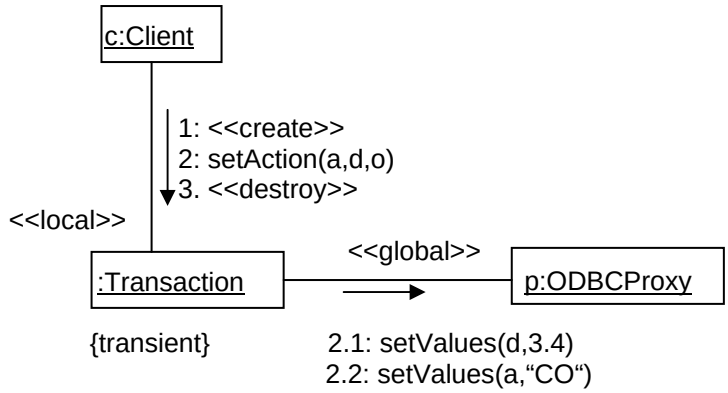


Abb.: Kolloborationsdiagramm

Zustandsdiagramm

Ein Zustandsdiagramm zeigt einen Automaten, der aus Zuständen, Zustandsübergängen, Ereignissen und Aktivitäten besteht. Bei Zustandsdiagrammen steht das nach Ereignissen geordnete Verhalten eines Objekts im Vordergrund.
Zustände werden symbolisch über Rechtecke mit abgerundeten Ecken dargestellt. Durch das Eintreffen von Ereignisssen kann ein anderer Zustand erreicht werden, was durch Pfeile angedeutet wird. Die Pfeile haben eine Beschriftung für das auslösende Ereignis und stellen Übergänge dar. In den Zuständen werden Operationen mit einer gewissen Zeitdauer ausgeführt, die Aktivität genannt werden. Im Gegensatz dazu wird die Zeitdauer von Aktionen mit Null angenommen. Aktionen sind Operationen, die an Übergängen ausgeführt werden. Angezeigt werden Aktionen durch einen Schrägstrich vor dem Namen.

Selbstverständlich können Übergänge auch an Bedingungen, die in eckigen Klammern stehen, geknüpft werden.

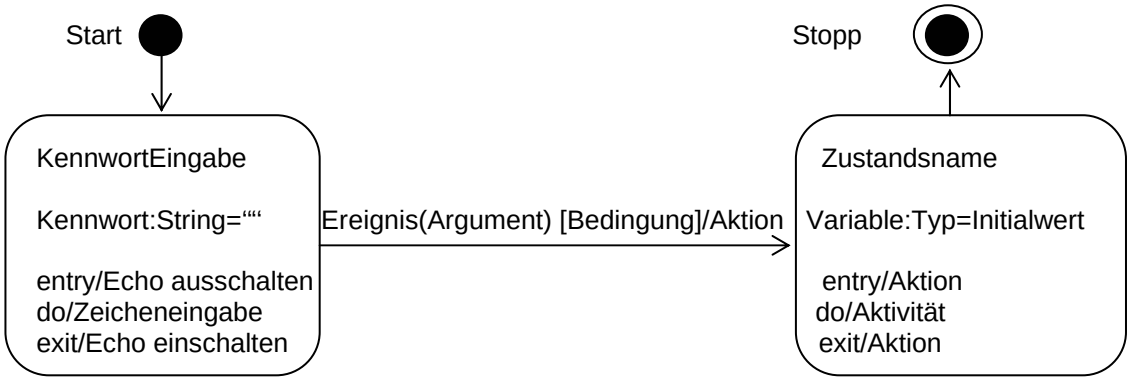


Abb.: Symbole im Zustandsdiagramm

Die Wörter `entry`, `do`, `exit` sind reserviert und können als Bezeichnungen für Ereignisse nicht verwendet werden.

`entry` gibt an, welche Aktion im Hintergrundausgeführt wird.
`exit` gibt an, welche Aktion beim Verlassen des Zustands ausgeführt wird
`do` beschreibt den Aufruf eines verschaltelten Zustandsdiagramms. Allgemein bezeichnet `do` eine Aktivität, die (unendlich) lange dauern kann.

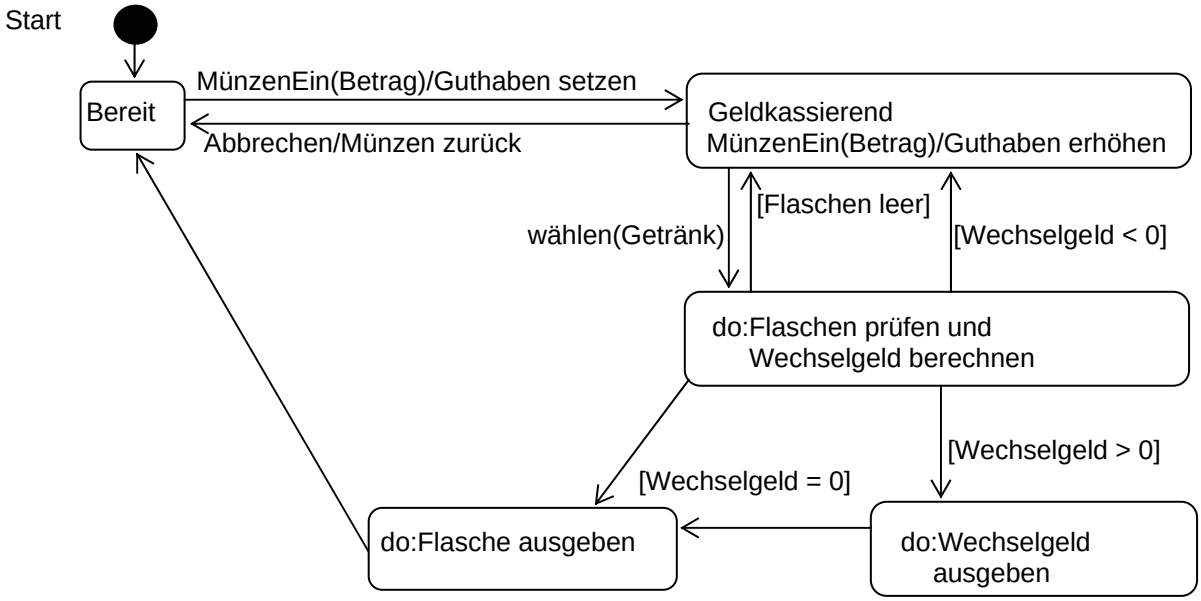


Abb.: Zustandsdiagramm für einen Getränkeautomaten

Aktivitätsdiagramm

Ein Aktivitätsdiagramm ist ein Sonderfall des Zustandsdiagramms, der den Fluß von einer Aktivität zu einer anderen innerhalb eines Systems zeigt. Eine Aktivität (actvity) ist ein andauernder, nichtatomarer Ablauf innerhalb eines Automaten. Aktivitätsdiagramme enthalten üblicherweise

- Aktivitätszustände und Aktionszustände
- Zustansübergänge (Transitionen)
- Objekte

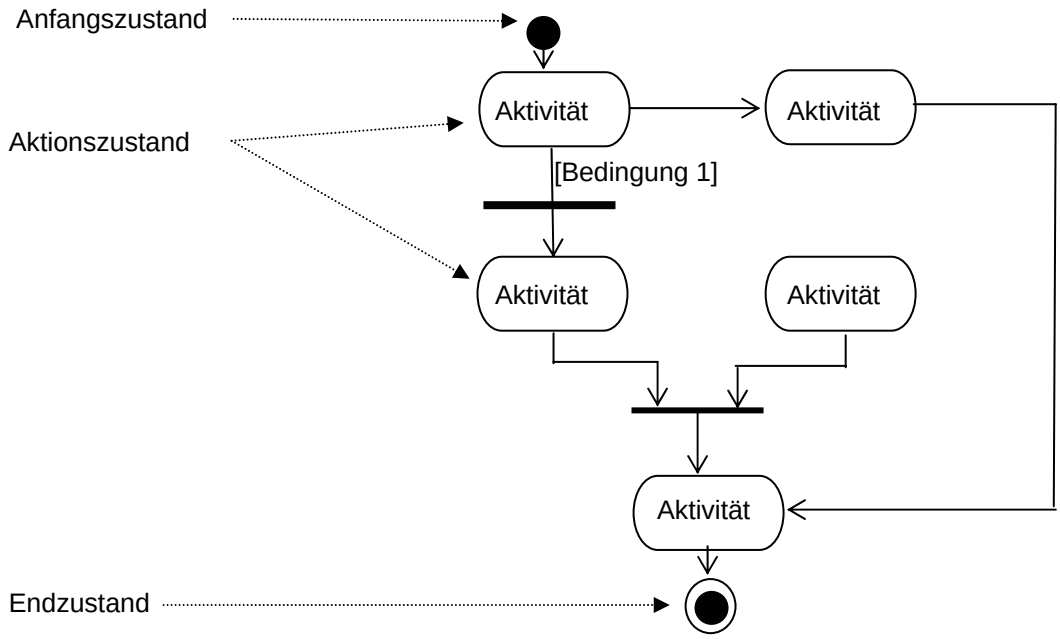


Abb.: Aktivitätsdiagramm

1.3.8.2 Schema-Modellierung

Die Klassendiagramme der UML bilden eine Obermenge von Entity-Relationship-Diagrammen. Klassendiagramme lassen auch das Modellieren von Verhaltensweisen zu. In einer physischen Datenbank werden diese logische Operationen im allg. in Trigger oder gespeicherte Prozeduren umgewandelt.

1. Modellierung eines logischen Datenbankschemas

Das Modellieren eines logischen Datenbankschemas[46] umfaßt:

- die Identifikation von Klassen, deren Zustand die Lebensdauer ihrer Anwendungen überdauern muß.
- das Erstellen eines Klassendiagramms, das all diese Klassen enthält, und das Markieren dieser Klassen mit den Standardeigenschaftswert `persistent`.
- Expansion der strukturellen Eigenschaften dieser Klassen (Spezifikation der Details zu den Attributen, Konzentration auf Assoziationen und Kardinalitäten).
- Beachtung typischer Muster, z.B. rekursive Assoziationen, Eins-zu-eins-Assoziationen, n-äre Assoziationen.
- Betrachtung der Vehaltensweisen dieser Klassen, z.B. durch Expansion von Operationen, die für Datenzugriff und Integrität der Daten wichtig sind.
- Überführung, falls möglich mit Hilfe von Programmen, des logischen in einen physischen Entwurf

[46] Booch, Grady u. a.: Das UML-Benutzerhandbuch, Addison-Wesley, Bonn 1999, Seite 123

Bsp.: Modellierung eines Datenbankschemas für eine Fachhochschule

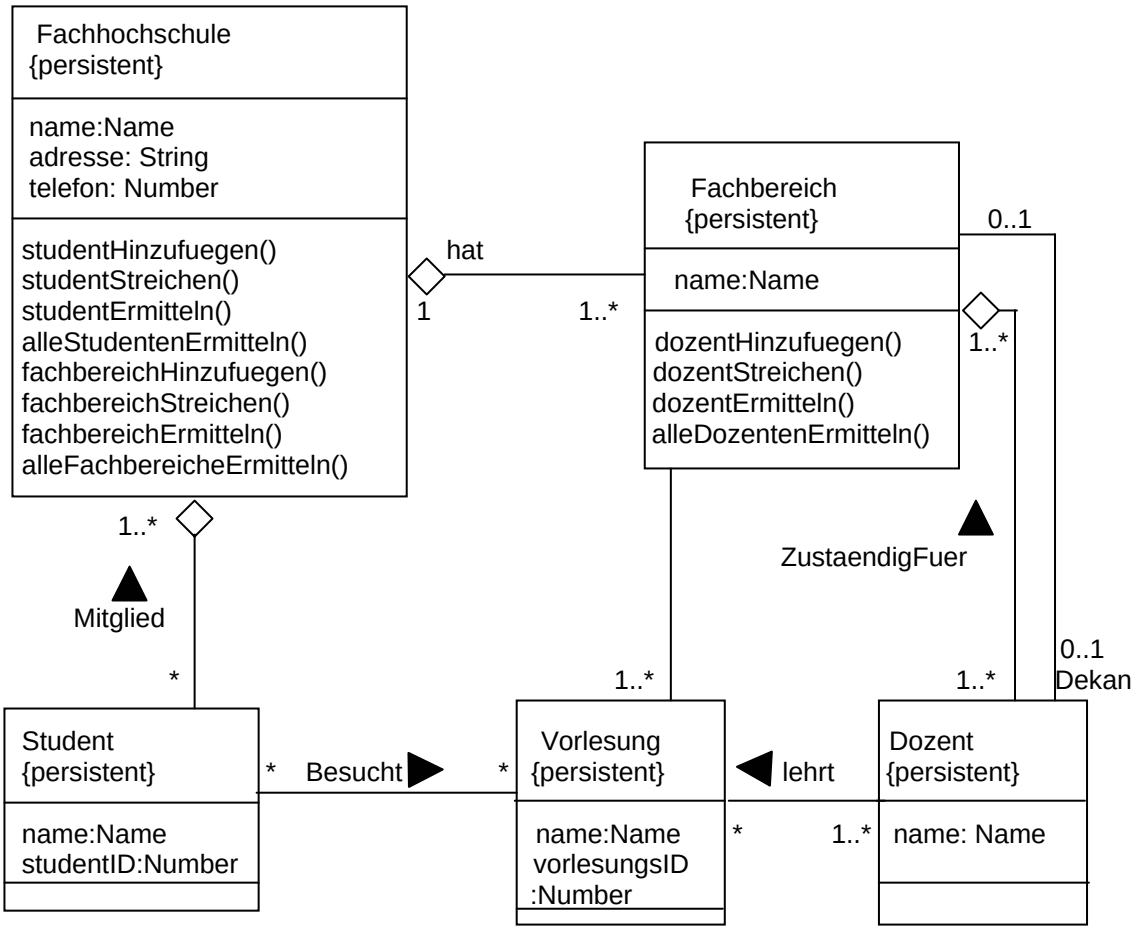


Abb.: Modellieren eines logischen Datenbankschemas

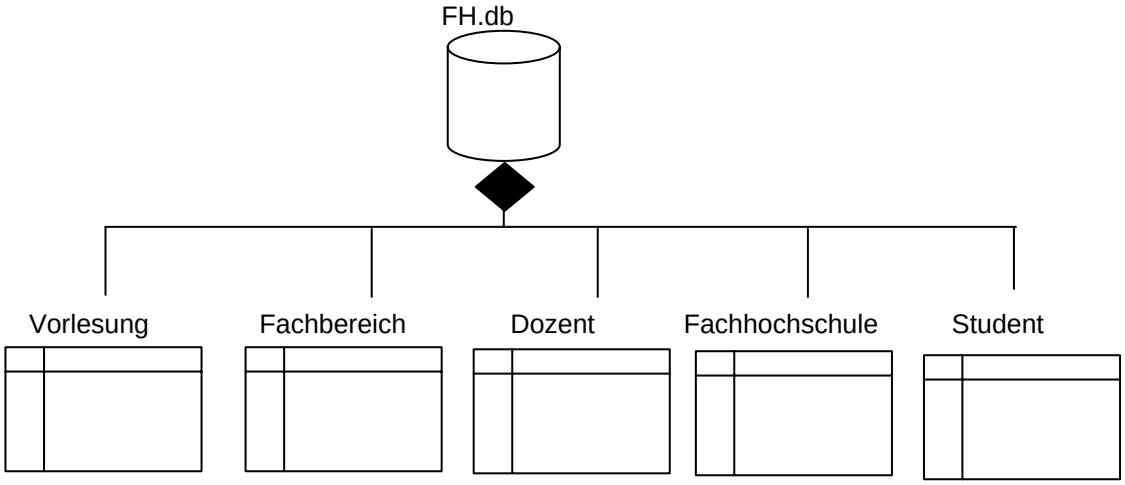
2. Die Modellierung eines physischen Datenbankschemas

Das Modellieren einer physischen Datenbank[47] umfaßt:

- Identifikation der Klassen, die das logische Datenbankschema bilden
- Wahl einer Strategie zur Abbildung dieser Klassen auf Tabbellen.
- Erstellen eines Komponentendiagramms (mit den als Tabellen stereotypisierten Tabellen) zur Visualisierung und Dokumentation

[47] vgl. Booch, Grady u.a.: Das UML-Benutzerhandbuch, Seite 451.

Bsp.: Modellieren der physischen Datenbank FH.db



1.3.8.3 Datenbankmodellierung mit der UML

Ausgangspunkt für das Modell sind **Use Cases**. Daraus lassen sich Struktur (stat. Modell) und Verhalten (dynamisches Modell) entwickeln. Die Modelle werden durch UML-Diagramme dargestellt und soweit verfeinert, dass eine Realisierung durch ein Datenbanksystem möglich ist.
Aus der Use-Case Beschreibung lassen sich fachliche Begriffe (Objekte), ihre Aufgaben, Eigenschaften und Zusammenhänge ermitteln. Objekte und ihre zugehörigen Klassen können in Form von Diagrammen dargestellt werden. **Klassendiagramme** modellieren die statischen Beziehungen zwischen Klassen bzw. Objekten.
Die Interaktionen mit den Use Cases lösen Transaktionen aus. Transaktionen stzen sich aus einer endlichen Folge von (elementaren) Operationen auf Objekten zusammen und können als **Seqenzdiagramm** übersichtlich dargestellt werden. Neben dem zeitlichen Ablauf der Aktivitäten sind auch die beteiligten Objekte sichtbar. Der Bedarf lässt sich der interne Ablauf der Methode durch ein **Aktivitätendiagramm** weiter präzisieren, so dass er als Methoden- und Protedurcode ins Datenbankschema aufgenommen werden kann.

1.3.9 Semistrukturierte Datenmodelle

Informationssysteme enthalten Daten aus Textdokumenten und HTML-Dokumenten des World Wide Web. Text- und HTML-Dokumente sind in der Regel nicht so stark strukturiert wie Datenbankdaten. Sie besitzen jedoch eine interne, oft wechselnde und nicht streng typisierte Struktur. Solche Daten bzw. Dokumente bezeichnet man als semistrukturiert.

Merkmale semistrukturierter Daten

- Die Struktur der Daten ist unregelmäßig
- Das Schema ist implizit in den Daten enthalten
- Die Struktur der Daten ist unvollständig
- Das Schema ist flexibel
- Das Schema ist relativ groß
- Das Schema unterliegt häufigen Änderungen Die Trennung zwischen Daten und Schema ist unscharf

Modelle für semistrukturierte Daten

Semistrukturierte Daten werden in der Regel mit Hilfe von graphbasierten Modelle beschrieben, z.B.

- Object Exchange Model (OEM)
- eXtensible Markup Language (XML)

1.3.9.1 Das Object Exchange Model

Bei dem OEM enthält das Modell Information zu:

- Label ist eine (klassenähnliche) Bezeichnung, der den Knoten einer Klasse gleichartigen Knoten zuordnet.
- Type kann ein atomarer Typ (z.B. string, int) oder ein mengenwertiger Typ sein
- Value ist ein zum Typ passender Wert.
- Object-ID der einzelnen Knoten

Jeder Knoten in einem Dokumentgraphen entspricht einem Objekt in einem Objektgraphen, Kanten entsprechen Referenzen bzw. Attributen vom Typ String.

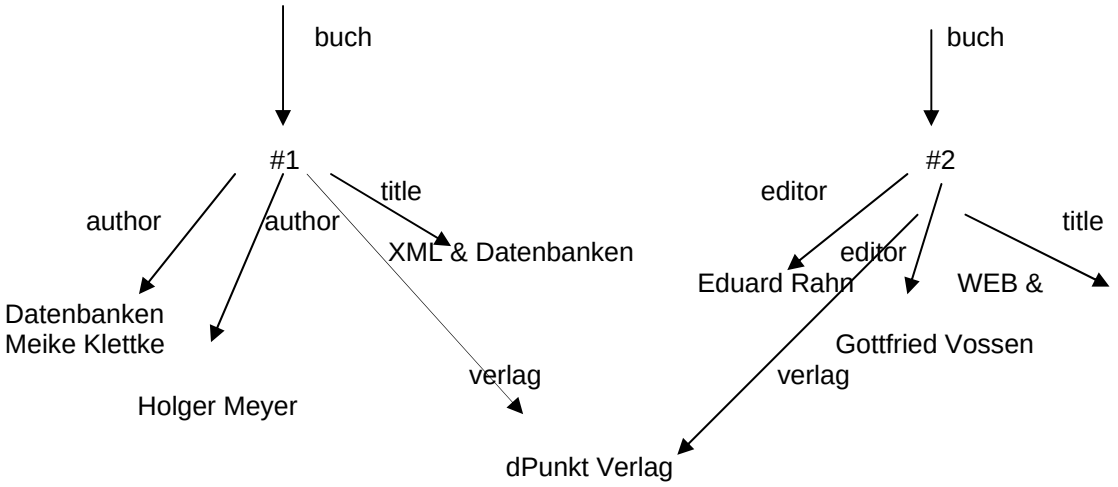


Abb. Darstellung von Daten in einem Dokumentgraph

Das OEM wird für die Semantikdefinition von Anfragesprachen verwendet. Für Datenbankbenutzer ist die Darstellung semistrukturierter Daten mit der XML von größerem Nutzen

```
<lexikon>
 <eintrag stichwort="Information">
  <herkunft>lat.</herkunft>
  <erklaerung num="1">Auskunft, Nachricht, Mitteilung, Belehrung</erklaerung>
  <erklaerung num="2"><siehe_auch>Fachinformation</siehe_auch></erklaerung>
  <erklaerung num="3"><anwendung>Informatik:</anwendung> die formulierte
    Unterrichtung nicht nur von Menschen, sondern auch von
    Organisationen und techn. Einrichtungen über Sachverhalte,
    Ereignisse, Abläufe. Die <siehe_auch>Informationstheorie
   </siehe_auch> versteht unter Informatik ein Maß, das den Zeichen einer
   Nachricht zugeordnet ist.
  </erklaerung>
  ...
 </eintrag>
</lexikon>
```

Abb.: XML-Dokument zur Darstellung semistrukturierter Informationen (Quelle der Inhalte: Der Brockhaus in fünf Bänden)

1.3.9.2 XML-Datenmodell

XML (eXtensible Markup[48] Language) ist eine standardisierte Syntax zur Beschreibung hierarchisch strukturierter Daten. XML ist keine Programmiersprache sondern ein universelles Format für strukturierte Dokumente und Daten. Die Tag-basierte Syntax ähnelt der von HTML (Hypertext Markup Language). Die HTML gibt einem Browser vor, wie die Daten anzuzeigen sind. XML teilt Applikationen mit, was die Daten bedeuten[49].
XML eignet sich vor allem zur Lösung von Austauschproblemen[50] zwischen heterogenen Systemen, XML-Daten sind leicht zugänglich, einfach zu konvertieren und abzuspeichern.

```
<?xml version="1.0" encoding="UTF-8"?>
<rechnung kundennummer="k333063143">
      <monatspreis>0,00</monatspreis>
      <einzelverbindungsnachweis>
            <verbindung>
                  <datum>26.2.</datum>
                  <zeit>19:47</zeit>
                  <nummer>200xxxx</nummer>
                  <einzelpreis waehrung="Euro">0,66</einzelpreis>
            </verbindung>
            <verbindung>
                  <datum>27.2.</datum>
                  <zeit>19:06</zeit>
                  <nummer>200xxxx</nummer>
                  <einzelpreis waehrung="Euro">0.46</einzelpreis>
            </verbindung>
```

[48] mark up: ursprünglich eine Anweisung aus dem Verlagswesen, Anweisung an den Setzer; definiert Regeln für den Aufbau von Dokumenten, die z.T. einer fest vorgegebenen Struktur entsprechen, teilweise aber auch Elemente beinhalten, die nicht diesem statischen Schema entsprechen (Wikipedia 16.3.2007)

[49] Daten und Informationen über Daten befinden sich in einem Dokument.

[50] Electronic Data Exchange (EDI) –Protokoll zur Übermittlung von Handelsdaten, neuerdings in XML-Kodierung

```
            <verbindungskosten_gesamt waehrung="Euro">2.19
            </verbindungskosten_gesamt>
      </einzelverbindungsnachweis>
</rechnung>
```

XML-Daten weisen eine stark verzweigte Baumstruktur auf.

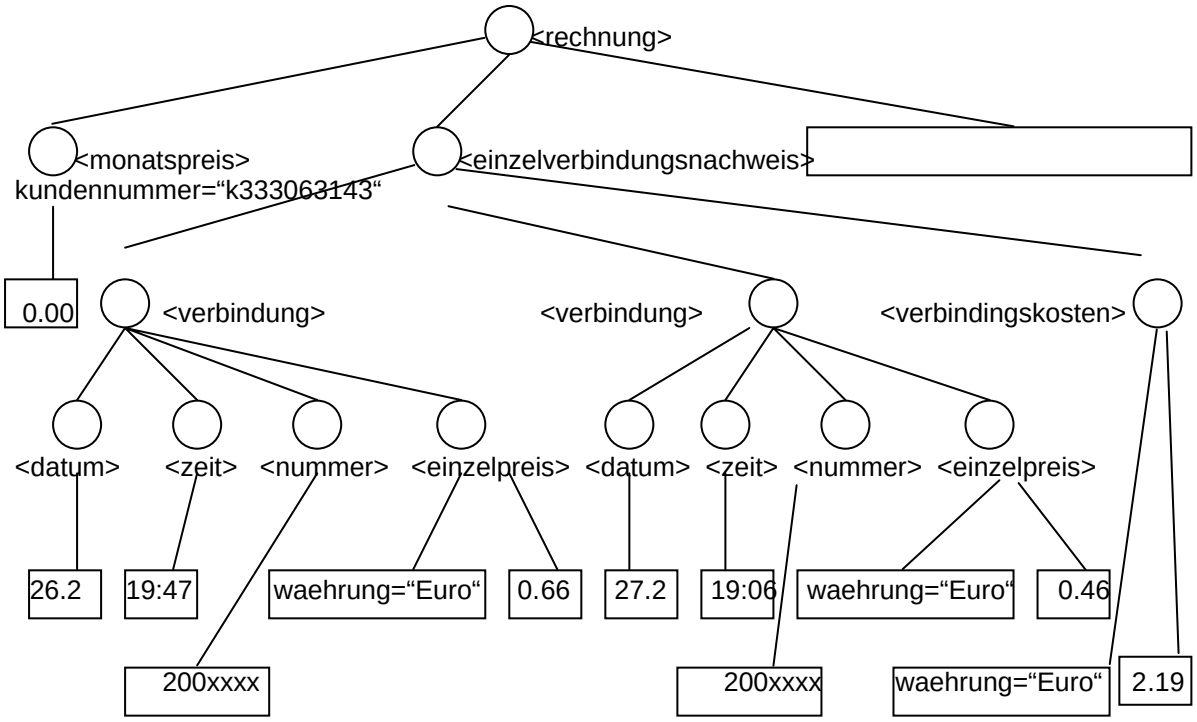


Abb.

XML-Daten weisen ein stark verzweigte Baumstruktur auf. Die Struktur kann durch Analyse (Parsen) von XML-Dokumenten bearbeitet werden.

1.3.9.2.1 XML-Dokumentenstruktur

Bestandteile eines XML-Dokuments

- Prolog (Vorspann)	`<?xml version="1.0" encoding = 'UTF-8'?>`
-- XML-Deklaration	
-- DTD[51]	`<!DOCTYPE beispiel1 SYSTEM "beispiel1.dtd" >`
-- Verarbeitungsanweisungen[52]	`<? PI-Name PI-Anweisung ?>`
- Kommentare	`<!-- Kommentar -->`
- Root-Element	`<skriptum>`

Die erste Zeile ist eine XML-Deklaration, die mit <? beginnt und mit ?> endet.
`<?xml version="1.0" encoding = 'UTF-8'?>`: Das XML-Dokument ist hier nicht „standalone“[53], d.h. es besitzt eine zugehörige DTD, die in der nächsten Zeile angegeben ist.

[51] DTD: liefert die Definition und die Beziehungen zwischen den in XML-Dokumenten enthaltenen Elementen
[52] Processing Instruction (PI), die Anwendung muß die PI auswerten (z.B. mit XML-Prozessoren)
[53] 3. Attribut im Prolog

Komponenten von XML-Dokumenten

- **Entity**: In XML-Terminologie werden Grundelemente von XML-Dokumenten als Entitäten bezeichnet, die entweder verarbeiteten Text (`parsed character data`) oder Rohtext (`unparsed character data`) enthalten. Der verarbeitete Text umfaßt den Dateninhalt und auch deklarative Textauszeichnungen, die die logische Struktur des Texts beschreiben.
- **Element**: Der Grundbaustein eines XML-Elements umfasst ein Start-Tag, ein End-Tag und den Text dazwischen, z.B.: `<vorname>Jürgen</vorname>`. In XML werden die beschriebenen Daten zwischen applikationsspezifischen Tags oder „Elementen“ hinterlegt. Die Syntax dazu wird in speziellen Dokumenten, den sog. Dokumenttypdefinitionen (DTDs) beschrieben.
- **Tag**: Ein dem Element zugewiesener Name. Die Merkmale zur Auszeichnung der Dokumente heißen Tags. Die syntaktische Definition erfolgt entweder mit ihrem erstmaligen Auftreten im Dokument oder durch formale Deklaration der (verwendeten) XML-Elemente in der DTD. Tags werden zur bedeutungsorientierten (semantischen) Beschreibung eingesetzt.
- **Attribut**: Ein Textname und ein Wertepaar, die zum Start-Tag hinzugefügt werden, um zusätzliche Informationen über das Element bereitzustellen.

Der Grundbaustein eines XML-Dokuments ist ein **Element**. Ein Element besitzt ein Beginn- und ein Ende-Tag, welche Daten beliebigen Inhalts einschließen. Durch die Schachtelung von Elementen erhält man hierarchische Strukturen.

```
<Name> Inhalt </Name> bzw. <Name/>[54]
```

Weitere Eigenschaften können durch **Attribute** definiert werden. Attribute werden in den Beginn-Tag der Elementdefinition eingeführt. Es dürfen keine gleichnamigen Attribute in einem Element auftreten.

```
<Name AttributName1='Wert1' .. AttributNameN='WertN'> ..
```

Jedes XML-Dokument instantisiert ein Infoset (ein abstraktes Datenmodell), das aus einem Dokument und seinen Informationsobjekten besteht. Informationsobjekte sind meistens Elemente, Attribute und Inhalte, z.B.:

```
<skriptum>
<title> Datenbanken </title>
<title> Typologie der Datenbanksysteme </title>
 <text> &kap1; </text>
</kapitel>
</skriptum>
```

Die Tags <skriptum> und </skriptum> definieren die Start- und Enpunkte. Das XML-Dokument beschreibt das Skriptum zur Vorlesung Datenbanken. Das erste Kapitel des Skriptums besitzt die Bezeichnung „Topologie der Datenbansysteme“ und ist nicht innerhalb des XML-Dokuments gespeichert. Ein Zeiger auf die externe Datei ist zu diesem Zweck angegeben. „`title`“ ist ein Tag und ein Element. Der Inhalt der Elemente kann durch Werte zwischen Tags und innerhalb von Tags bereitgestellt werden. Elemente können Inhalte, Child Elemente oder beides enthalten, oder sie können leer sein. Zur Anlage eines leeren Elements hat das zugehörige Tag folgendes Format: </name>.
Elemente besitzen auch Attribute, z.B. hat das Kapitel-Element eine Attribut `nummer`. Der Attributwert steht in doppelten Anführungszeichen: `<kapitel nummer=“1“>`
Bei Elementnamen und Attributnamen wird auf Groß- und Kleinschreibung geachtet.

[54] kann verwendet werden, falls der Elementinhalt leer ist.

Abb. Körper eines XML-Dokuments

Jedes Element besitzt exakt einen Parent, ein Parent kann eine beliebige Anzahl Kinder (bzw. children) besitzen. Da XML strikt hierachisch ist, muß beim Einfügen eines Elements innerhalb eines anderen Element das Ende-Tag des neuen Elements vor dem Ende-Tag des Parent-Elements stehen.
XML-Dokumente müssen bestimmte Syntaxregeln einhalten, z.B.:

Für jedes von Syntaxregeln definierte und verwendete öffnende Tag muß es ein schließendes Tag geben.

```
<?xml version=“1.0“ ?>
 <firstTag>
  <secondTag>
   <informationTag>information</informationTag>
  </secondTag>
 </firstTag>
```

Die Syntaxregeln sorgen für die hierarchische Integrität von Dokumenten.
Bsp.: Elemente und Attribute

```
<?xml version=“1.0“ ?>
 <angestelltenliste>
  <angestellte>
  <angid>A01 </angid>
  <name>Fritz</name>
  <gehalt>24000</gehalt>
  </angestellte>
 <angestelltenliste>
```

1 Elemente

```
<?xml version=“1.0“ ?>
 <angestelltenliste>
  <angestellte angid=“A01“
               name=“Fritz“
               gehalt=24000
  </angestellte>
 </angestelltenliste>
```

2 Attribute

Wohlgeformte und gültige Dokumente

Ein "wohlgeformtes XML-Dokument" muß folgende Syntaxregeln einhalten:

- Das Dokument muß mit der XML-Deklaration beginnen
- Das Dokument muß über ein sog. Root-Element verfügen, das alle übrigen Elemente enthält. (
- Alle Elemente müssen über ein Start- und ein End-Tag verfügen, es sei denn, es handelt sich um leere Elemente
- Leere Elemente müssen nur über ein Tag im Format <name/> verfügen
- Attributwerte müssen in Anführungszeichen gesetzt werden.

Auch wenn ein XML-Dokumente diesen Regeln entspricht, muß es noch nicht gültig sein. Das XML-Dokument muß auch noch den Regeln einer Dokumenttypdefintion (**DTD**) entsprechen. Die DTD beschreibt, welche Elemente und Entitäten in einem XML-Dokument erscheinen, und wie die Inhalte und Attribute des Elements aussehen.
Zur maschinellen Verarbeitung von XML-Dokumenten dient ein Verarbeitungsprogramm, ein sog. XML-Prozessor, der die syntaktische Korrektheit

und den korrekten Aufbau eines XML-Dokuments überprüft. Deshalb wird in XML zwischen wohlgeformten und gültigen Dokumenten unterschieden.
Gültige XML-Dokumente entsprechen nicht nur den syntaktischen Regeln von XML, sondern enthalten ausschließliche gültige Markierungen, die anwendungsspezifisch definiert werden.

1.3.9.2.2 XML-Standards

DTD (Dokumententypdefinition)

Dokumenttypdefinitionen spezifizieren, wie XML-Elemente, Attribute und andere Daten definiert sind und mit welchem logischen Bezug zueinander sie innerhalb eines Dokuments verwendet werden können. Die DTD unterstützt Definition und Beziehungen zwischen den in XML-Dokumenten enthaltenen Elementen. Die Dokumenttypdefinition (document typ declaration, **DTD**) legt die Struktur gültiger XML-Dokumente fest, d.h.: in welcher Reihenfolge die Auszeichnungen in den XML-Dokumenten auftreten dürfen, und wie diese geschachtelt sein dürfen.

- DTDs gehen in die Struktur von XML-Dokumenten ein.
- DTDs können in einer separaten Datei außerhalb von XML-Dokumenten eingebunden werden.
- Bei DTDs handelt es sich um Sätze von Definitionen, die von der XML-Engine interpretiert werden, um Dokumente auf Gültigkeit zu überprüfen.
- Alles, was nicht ausdrücklich in einer DTD erlaubt ist, ist verboten.

Die Definition der gültigen Auszeichnungen mit Hilfe der DTD kann entweder innerhalb eines XML-Dokuments (intern) oder durch eine externe Dokumenttypdefintion erfolgen, auf die das XML-Dokument verweist

```
<?xml version=“1.0“ encoding=“UTF8“ ?>
<!DOCTYPE Adresse SYSTEM ,“…/dtds/adresse.dtd“>
<Adresse>
 <Name> Christian Ditzel </Name>
 <Strasse> Kirchplatz 7 </Strasse>
 <Stadt> Bad Hersfeld </Stadt>
 <Postleitzahl> 36251 </Postleitzahl>
 <Land> Deutschland </Land>
</Adresse>
```

Die erste Zeile bezeichnet die XML-Version.
Die zweite Zeile ist die Dokumentypdeklaration (beginnt mit <!DOCTYPE …), die den Dokumenttyp (hier: Adresse) festlegt und auf eine Datei verweist, die Deklaration der Elemente und Attribute enthält. Jedes gültige XML-Dokument enthält ein Wurzelelement, das den gleichen Namen, wie der zugrundeliegende Dokumenttyp trägt. Das Wurzelelement umschließt den ganzen Inhalt des Dokuments. Eine DTD muß in ihrem Namen den Namen des Wurzelelements enthalten.

Abb.: Einfaches Bsp. Für ein XML-Dokument

Bei Verwendung einer internen Dokumenttypdefinition erfolgt kein Verweis auf eine externe Datei, die notwendigen Deklarationen werden innerhalb des Dokuments angeführt. Ein XML-Element wird mit Hilfe einer Elementdeklaration definiert, die mit `<!ELEMENT` eingeleitet wird. Danach folgt der Name des Elements und er Wertebereich des Inhalts. Falls hier `#PCDATA` angegeben ist,

dann darf der Wertebereich keine Zeichen umfassen, die zur Auszeichnung vorgesehen sind[55].

```
<?xml version=1.0 encoding='UTF8' ?>
<!DOCTYPE Adresse [
 <!ELEMENT Adresse (Name,Strasse,Stadt,Postleitzahl,Land) >
 <!ELEMENT Name (#PCDATA)>
 <!ELEMENT Strasse (#PCDATA)>
 <!ELEMENT Stadt? (#PCDATA)>
 <!ELEMENT Postleitzahl (#PCDATA)>
 <!ELEMENT Land (#PCDATA)>
 ]>
<Adresse>
 <Name> Christian Ditzel </Name>
 <Strasse> Kirchplatz 7 </Strasse>
 <Stadt> Bad Hersfeld </Stadt>
 <Postleitzahl> 36251 </Postleitzahl>
 <Land> Deutschland </Deutshland>
</Adresse>
```

Die Zeile `<!ELEMENT Name (#PCDATA)>` deklariert das XML-Element mit der Bezeichnung Name innerhalb des XML-Dokuments. Durch `(#PCDATA)` wird festgelegt, daß der Inhalt des Elements Name verarbeiteter Text (Parsed character data) ist. Das besagt: keine Unterelemente sind in diesem Element erlaubt.
Die Zeile `<!Element Adresse (Name,Strasse,Stadt,Postleitzahl,Land) >`definiert, daß das XML-Element Adresse aus den XML-Elementen Name, Strasse, Postleitzahl, Land zusammengesetzt sind. Diese Elemente sind innerhalb von Adresse geschachtelt. Das Fragezeichen nach Stadt besagt, daß dieses Element optional ist.

Abb. XML-Dokument mit interner Dokumenttypdefinition

XML-Dokumente können ohne DTD nur auf Wohlgeformtheit gemäß der XML-Spezifikation überprüft werden, sie werden bei Korrektheit als „well-formed“ bezeichnet. Genügt ein Dokument dagegen auch seiner DTD ist es gültig („valid“)[56].
Die in einem Dokumenttyp gültigen Elemente und deren Schachtelungsstruktur werden durch Verwendung von Elementdefinitionen (mit `<!ELEMENT … >` angegeben. Ebenso können die gültigen Attribute durch Attributdeklarationen (mi `<!ATTLIST … >` definiert werden.
Eine DTD ist kein Datenbankschema.

XML-Schema (Sprache zur Modellierung strukturierter Informationen)[57]

XML- Schema (oder XML Schema Definition, XSD) ist eine weitere Möglichkeit zur Definition einer Vorlage für XML-Dokumente[58]. Es hat ähnliche Funktion wie die DTD, unterstützt jedoch Datentypen und Namensräume.

- W3C-Standard zur Festlegung von Inhalt und Struktur von XML-Dokumenten
- Anbieter einer Vielzahl von Datentypen, Voraussetzung für strukturierte Speicherung
- Voraussetzung für strukturierte Speicherung, Definition eigener Datentypen, Bezugnahme auf vielfältige, vordefinierte Datentypen
- Das XML-Schema wird in in XML geschrieben

[55] Diesbezügliche Zeichen wie ‚<' und ‚>' sind somit in einem #PCDATA-Bereich nicht erlaubt und müssen in dem Element symbolisch durch Angaben wie `<` und `>` ersetzt werden.
[56] Gültigkeitsprüfung durch Bereitstellen einer kontextfreien Grammatik für Dokumente und deren Elemente
[57] W3C-Standard zur Festlegung von Inhalt und Struktur von XML-Dokumenten
[58] alternativ zur Dokumenttypdefinition der DTD

- sehr gute Unterstützung regulärer Ausdrücke

Der wohl wichtigste Unterschied zwischen der Definition mit Hilfe von XML-Schema und einer DTD ist, daß bei Verwendung des XML-Schema-Standards generell für jedes Element und Attribut der gültige Wertebereich festgelegt werden kann.

XSL (eXtensible Stylesheet Language)

Das XML-Format enthält keine Informationen darüber, wie die Daten dargestellt werden sollen. Dafür sind Style Sheets oder Skripte zusttändig.
XSL kann verwendet werden, um XML-Dokumente umzuwandeln oder zu formatieren. Insgesamt besteht XSL aus 3 Teilen: das eigentliche XSL, XSLT (XSL Transformations) und XPath (XML-Adressierungssprache).
XSLT (eXtensible Style Sheet Language Transformation) ist eine Sprache zur Transformation (Umwandlung) von XML-Dokumenten, die vom W3C standardisiert wird. Hierbei wird ein XML-Dokument gemäß der über XSLT definierten Regeln umgeformt. Die XSLT-Regeln werden in XML formuliert und ergeben ein XSLT-Dokument. Die umzuformenden Betandteile des Ausgangsdokuments werden von XSLT über XPath-Ausdrücke referenziert.

Namensräume

XML-Dokumente können aus Elementen von unterschiedlichen Dokumenttypdefinitionen zusammengesetzt werden. Da jede Dokumenttypdefinition beliebige Bezeichner (bzw. Elementnamen) vergeben kann, ist es möglich, daß in unterschiedlichen Schemadefinitionen der gleiche Name verwendet wird. Sollen diese Namen gemeinsam verwendet werden, kommt es zu einem Namenskonflikt.
Unter einem XML-Namensraum versteht man einen durch ein Präfix gekennzeichneten Bereich von Elementen. Durch die Definition eines Namensraums wird in einem XML-Dokument ein Präfix je Dokumenttypdefinition gewählt, die im XML-Dokument vor den entsprechenden Element- und Attributnamen gestellt werden.
Namensräume unterscheiden Elemente mit identischen Namen aus unterschiedlichen XML-Anwendungen.

XPath

XPath liefert die Syntax für das Durchsuchen von XML-Dokumenten, kann als Basis für Such- und Abfragesprache verwendet werden. Mit Hilfe von XPath kann man mit einer Pfad-Notation auf einzelne Knoten eines XML-Dokuments zugreifen

- aktuelle Version 1.0, 1999 vom W3C empfohlen
- Bestandteil von XQuery, XSL, Klink/XPointer
- Ausgabgspunkt ist die Baumstruktur eines XML-Dokuments
- liefert eine Knotenmenge zurück
- entspricht nicht XML-Syntax

XPath ist eine Abfragesprache für XML-Dokumente, über die Elemente mit angeführten Eigenschaften aus XML-Dokumenten herangezogen werden können. Die Bezeichnung XPath geht darauf zurück, daß Abfragen in Form eines (Teil-)

Pfads[59] von der Wurzel zu den gesuchten Elemente herangezogen werden können. Für jeden Teilausdruck entlang des Pfads können Einschränkungen (constraints) angegeben werden, die die Lösungswege einschränken. Anfragen werden als `for … let … order by … return …`-Ausdrücke (FLOWR[60]-Ausdrücke) formuliert. FLOWR-Ausdrücke können beliebig tief geschachtelt sein.

XQuery

Basiert auf XPath und erlaubt flexible Abfragen auf XML-Datenbanken. XQuery enthält auch Elemente prozeduraler Sprachen.

1.3.9.3 Abbildung von XML-Strukturen auf Datenbankstrukturen

Grundsätzlich kann man zwischen der Template-gesteuerten und der Modell-gesteuerte Abbildung unterscheiden.

Template-gesteuerte Abbildung

Es gibt hier keine vordefinierte Zuordnung zwischen der Struktur des XML-Dokuments und der Datenbank-Struktur. Statt dessen werden entsprechende Kommandos in ein Template eingebettet, die dann von der für den Datentransport verantwortlichen „Middleware“ verarbeitet werden. XML-Dokumente dienen hier als Templates (Schablonen) mit speziellen eingebetten Anweisungen, z.B. SELECT-Statements. Templates werden eingesetzt, um gezielt spezielle Informationen, z.B. aus einer SQL-Datenbank in ein XML-Dokument zu übertragen.

```
<?xml version=1.0“?>
<Fluginformationen>
 <Intro> Folgende Flüge haben noch freie Plätze:</Intro
 <SelectStmt> SELECT Airline, FltNumber, Depart, Arrive FROM Flights
</SelectStmt
</Fluginformation>
```

Abb.: XML-Template mit SELECT-Anweisung

Das Template wird bspw. von einem Server bearbeitet:

- Herausfiltern der SELECT-Anweisung nach dem Parsen
- Weiterleitung an die Datenbank
- Ergebnisrelation in XML-Format bringen („XML-Middleware“)
- Anzeigen

Template gesteuerte Abbildungen werden derzeit nur für die Überführung von Daten aus relationalen Datenbanken in XML-Datenbanken genutzt.

Modell-gesteuerte Abbildung

[59] Für Pfadausdrücke gibt es die vom World Wide Web Consortium standardisierte Sprache XPath, die u.a.a für XML Style Sheets verwendet wird.
[60] Ausgesprochen: flower

Bei der Modell-gesteuerten Abbildung gibt es ein konkretes Modell nach dem die XML-Struktur implizit oder explizit auf die Datenbank-Struktur[61] abgebildet wird. Zwei bekante Modelle sind das Table Modell, bei dem ein XML-Dokument auf einzelne oder mehrere Tabellen einer relationalen Datenbank abgebildet wird, und das datenspezifische Objekt-Modell, bei dem der XML-Baum direkt auf eine objektorientierte oder hierarchische Datenbank abgebildet wird.

Bsp.: Table-Modell

XML-Dokument-Struktur entspricht dem relationalen Schema (oder Teilen davon) bzw. einer Ergebnismenge

```
<database>
 <table>
  <row>
   <column1> … </column1>
   <column2> ... </column2>
   …
  </row>
 </table>
</database>
```

Abb.: Kanonische Abbildung auf <db>- <table>- <attribut> - Tags

Die XML/DB-“Middleware” füllt die Elemente oder entnimmt die Daten. Die Abbildung der Daten erfolgt auf Meta-Elemente der Datenbank (Tabellen, Spalten, Views).

Bsp.: Objekt-Modell
Der Objektbaum (tree) eines XML-Dokuments stellt die Eingabereihenfolge der von der zugehörigen DTD definierten Elemente der XML-Struktur übersichtlich dar.

```
<?xml version = “1.0“ ?>
<orders>
 <purchaseorder ID = “007“>
  <Customer>
   <name> Müller </name>
   <adress> … </adress>
  </Customer>
price
  <lineItem>
   <no> 4711 </>
   …
  </lineItem>
  <lineItem>
  …
  </lineItem>
</orders
```

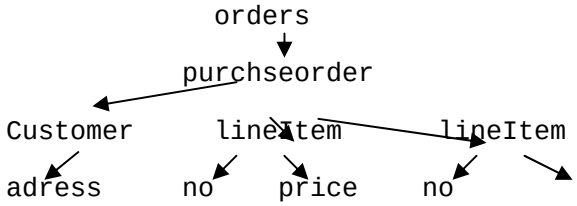


Abb. XML-Fragment mit datenspezifischem Objektbaum

Bei dieser Abbildung werden Element zu Objekten und Subelemente sowie Attribute werden zu Eigenschaften von Objekten. Das so entstandene Modell kann direkt auf eine objektorientierte oder eine hierarchische Datenbank abgebildet werden.

[61] oder auch in umgekehrter Richtung

1.3.9.4 Abbildung von XML-Daten auf Datenbanken (und umgekehrt)

Bei der Auswahl eines DBS zur Speicherung von XML-formatierten Informationen spielen Struktur und Inhalt der XML-Dokumente eine Rolle. Es gibt keine optimale Lösung für alle Anwendungen, der Dokumentcharakter spielt eine entscheidende Rolle.
Bei der Darstellung auf der **logischen Ebene** läßt sich der Wunsch nach einer Kombination von XML und Datenbanken je nach XML-Dokumenttyp folgendermaßen charakterisieren:

- datenzentriert: Daten(bank)modell, SQL, OQL, XPath, XQuery, ...
- dokumentzentriert: Dokumentmodell, XML, SGML, XSLT, XPath, XQuery, IR-Anfragen, DOM, ...

Bei der Darstellung auf der **physischen Ebene** spielen je Dokumenttyp eine Rolle:

- datenzentriert: Datenbankstrukturen wie Hashing, B-Bäume
- dokumentzentriert: Dateien, Volltextindex, invertierte Listen, Graphen (DOM), ...

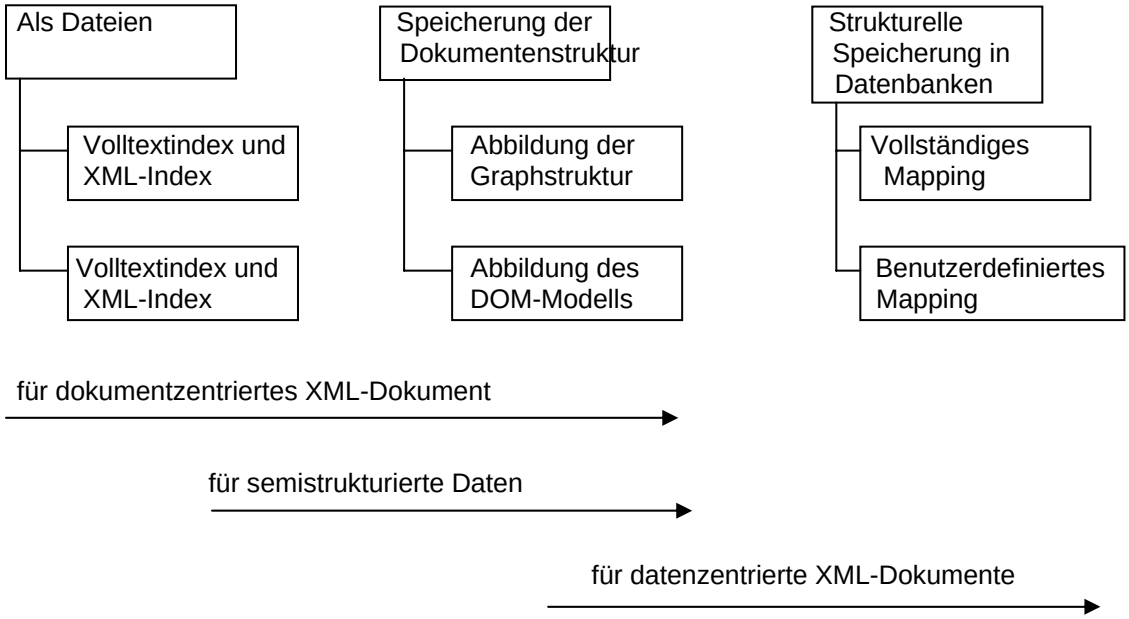


Abb.: Abbildung von XML-Dokumente

Zur **Abbildung Datenbanken in XML-Daten bzw. XML-Daten in Datenbanken** haben mehrere Abbildungsvarianten ihre Berechtigung:

textbasiert
- als Zeichenkette
- gut für dokumentzentrierte XML-Dokumente

graphbasiert
- als Graph (DOM)
- gut für semistrukturiert

strukturbasiert
- in Relationen
- gut für datenzentriert

Die **textbasierte Speicherung**

- kennt die Varianten: Datei, CLOB, spezieller XML-Textdatentyp
- führt sequentielle Anfragebearbeitung aus
- fordert den Einsatz spezieller Indexe:

-- Volltextindex wie in IR, etwa invertierte Listen
-- Pfadindex (Index auf Elemente/Attribut-Struktur)

Eine Bewertung der textbasiereten Speicherung ergibt:

- Zugriff relativ aufwändig
- keine Schemainformation nötig
- keine aufwändige Abbildung
- ordnungserhaltend
- keine SQL-ähnlichen Anfragen möglich

Die **graphbasierte Speicherung** kennt die Varianten

- einfache Speicherung der Graphenstruktur
- Document-Object-Modell-Speicherung (DOM)

Die **strukturbasierte Speicherung**

- führt zur Trennung von Metadaten (SQL-Schema) und Daten (Tupel) anhand DTD/Schema
- unter Ausnutzung objektorientierter/objektrelationaler Strukturierungskonzepte
- unterscheidet automatische Abbildung und nutzerdefinierte Abbildung

Eine Abbildung würde im Rahmen der strukturierten Speicherung umfassen:

- XML-Element in Attribut einer Relation
- Sequenz von Elementen in Attribute einer Relation
- Alternative von Elementen in Attribute einer Relation
- optionales Element (?) in Attribut mit evtl. Nullwert
- Elementwiederholungen (*,+) in Set oder List
- XML-Attribut in Attribut einer Relation mit evtl. Nullwert
- #required in not null

Bsp.: Die zuerst angegebene DTD und die danach durchgeführte Abbilung in eine Datenbank-Tabelle

```
DTD:
<!ELEMENT hotel (hotelname, kategorie,
adresse, telefon+, fax)>
<!ELEMENT hotelname (#PCDATA)>
<!ELEMENT kategorie (#PCDATA)>
<!ELEMENT adresse
(plz,ort,strasse,nummer)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT strasse (#PCDATA)>
<!ELEMENT nummer (#PCDATA)>
<!ELEMENT telefon (#PCDATA)>
<!ELEMENT fax (#PCDATA)>

SQL-2003:
CREATE TABLE hotel(
hotelname VARCHAR(50) NOT NULL,
kategorie INTEGER NOT NULL,
adresse ROW (
plz INTEGER NOT NULL,
ort VARCHAR(50) NOT NULL,
strasse VARCHAR(30) NOT NULL,
Nummer INTEGER NOT NULL
) NOT NULL,
telefon VARCHAR(30) MULTISET
NOT NULL,
fax VARCHAR(30)
```

```
);
```

Die Abbildung funktioniert nicht immer. Es gibt Probleme, z.B.:

- Abbildung von DTD auf SQL-Datentypen
- Welche Elemente werden in welche Relation gruppiert? (Normalisierung)
- Abbildung von DTD-Rekursion in separate Relationen
- Erzeugung Primär/Fremdschlüssel
- Ordnungserhaltung durch lfd. Nummer als künstliches Attribut
- Abbildung von Mixed Content
- Wurzelelemente in XML nicht festgelegt
- viele Nullwerte möglich

1.4 Standards

Auf dem DB-Markt existieren Systeme verschiedener Hersteller, die in ihren Strukturen, ihrer Terminologie und vor allem in den Sprachen erhebliche Unterschiede aufweisen. Schrittweise wurden von verschiedenen Gremien (DIN, ANSI, CODASYL, ISO) allgemeine DB-Standards empfohlen und Normenvorschläge unterbreitet. Die Empfehlungen weisen teilweise bzgl. Funktionsumfang und Realisierungsaufwand erhebliche Unterschiede auf, sind aber dennoch bedeutende Eckpfeiler der DB-Geschichte.
Von großer Bedeutung sind auch die durch die Position des Marktführeres IBM verbreiteten "defacto"-Standards (z.B. IMS, DL/1 oder das System R).

1.4.1 CODASYL-Konzept

Die "Conference on Data Systems Languages (**CODASYL**)" ist ein Ausschuß von Herstellern und großen amerikanischen Anwendern, der sich um die Normung von Programmiersprachen bemüht. Auch im Bereich der Datenbanken wurde die CODASYL-Gruppe aktiv. Gegen Ende der 60er Jahre wurde die "Data Base Task Group (**DBTG**)" beauftragt, einen Standardisierungsvorschlag für das bereits weit auseinanderlaufende Feld der Datenbankentwicklung auszuarbeiten.
Die wichtigsten Empfehlungen der Ausarbeitung waren:

1. Definition der Syntax einer Datenbank-Beschreibungssprache (**DDL**)
2. Definition der Syntax einer Datenbank-Manipulationssprache (**DML**)

Darüber hinaus wurde in einer "Feature Analysis" ein Modell-DBMS konzeptiert. Dieses Konzept, von führenden DB-Fachleuten erarbeitet, steht allen Herstellern und Anwendern zur Verfügung. Seine externen Schnittstellen sind exakt beschrieben, die interne Realisierung bleibt jedem Hersteller selbst überlassen. CODASYL-DB werden bzw. wurden angeboten von Honeywell (IDS), Univac (DMS/1100), Cullinane Corp. (IDMS) und Siemens (UDS).

1.4.2 IMS

`IMS` (Integrated Mangement System) ist eines der ältesten Systeme und daher eines der am weitesten entwickelten DB/DC-Systeme. `IMS` wurde bereits Mitte der 60er Jahre konzipiert, die 1. Version Ende 1968 freigegeben. Seit 1973 ist IMS in den virtuellen Systemen (OS/VS1 und OS/VS2) verfügbar (IMS/VS). Darüberhinaus wurden am Markt die Varianten DL/1 DOS/VS und DL/1-Entry angeboten, die funktionell eine Untermenge von IMS/VS darstellen.

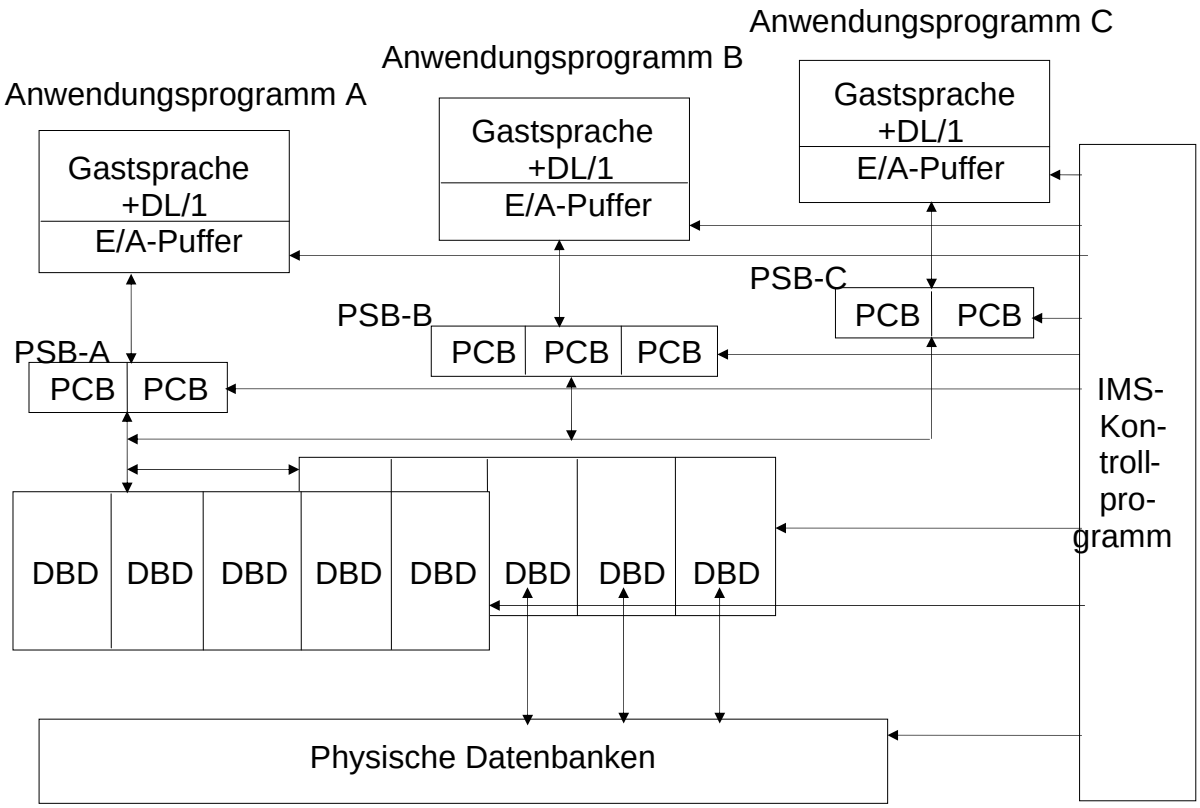


Abb. 1.4-1: Aufbau IMS

Kurzbeschreibung der DB-Struktur von IMS

IMS besteht in der Regel nicht aus einer einzigen, sondern aus mehreren Datenbanken. IMS erlaubt den Aufbau hierarchisch strukturierter Datenbestände, die für das System in 3 verschiedenen Stufen beschrieben werden. Man unterscheidet zwischen der Beschreibung des hierarchischen Datenbankaufbaus, der Beschreibung des Zugriffs für ein bestimmtes Programm und der Beschreibung der Segmente mit den Datenfeldern. Als Segment bezeichnet man die kleinste Zugriffseinheit, das ist die kleinste Menge an Daten, die durch DL/1-Operationen transportiert werden kann (DL/1 ist die Sprache (DDL, DML)) von IMS. Zur Beschreibung des hierarchischen Aufbaus einer physischen Datenbank dient die „Data Base Description (DBD)“. Eine physische Datenbank ist die tatsächlich gespeicherte Datenbank, im Gegensatz zum Datenbankausschnitt (program communication block, PCB), der dem Benutzer zur Verfügung gestellt wird. Die Beschreibung des speziellen Datenbank-Zugriffs für ein Programm erfolgt im "program specification block (PSP)". Felder und Struktur eines Segments werden im Programm selbst beschrieben, sie entsprechen den Konventionen der jeweiligen Wirtsprache.

1.4.3 System R und SQL

1.4.3.1 System R

Hierbei handelt es sich um den Prototyp[62] relationaler Datenbanken. Die Entwicklung erfolgte in den Forschungslabors der IBM. Aktuell davon ist vor allem der Sprachentwurf SEQUEL (**SQL**) bzw. die "zweidimensionale" Zugriffssprache "Query by Example (**QBE**)"[63]. Die beiden Sprachen sind grundlegend für den Sprachentwurf in relationalen Datenbanken.

1.4.3.2 Standard-SQL (Structured Query Language)

Im Februar 1987 wurde SQL zum offiziellen Standard des American National Standard Institute (ANSI). Im März 1987 wurde SQL die Datenbank-Zugriffssprache im Rahmen der System Application Architecture (SAA) von IBM. Inzwischen wurde der "Standard von 1986" bereits zweimal aktualisiert (1989, 1992). Die aktuelle Fassung (SQL-92, aus "taktischen Gründen" SQL2 genannt) umfaßt bereits eine 600 Seiten umfassende Beschreibung.

1. Die **`select`**-Anweisung

- Die zentrale Idee

Die zentrale Idee, die SQL zugrundeliegt und in der **select**-Anweisung verwirklicht wurde, ist eine Abbildung bestimmter Tabellenspalten. Die Tabellen bilden den Definitionsbereich. Über eine Auswahlkriterium wird aus dem Definitionsbereich ein Abbildungsbereich bestimmt:

```
select <spalten>            /* 1. Klausel */
from <tabellen>             /* 2. Klausel */
[where <bedingung> ... ]    /* 3. Klausel */
```

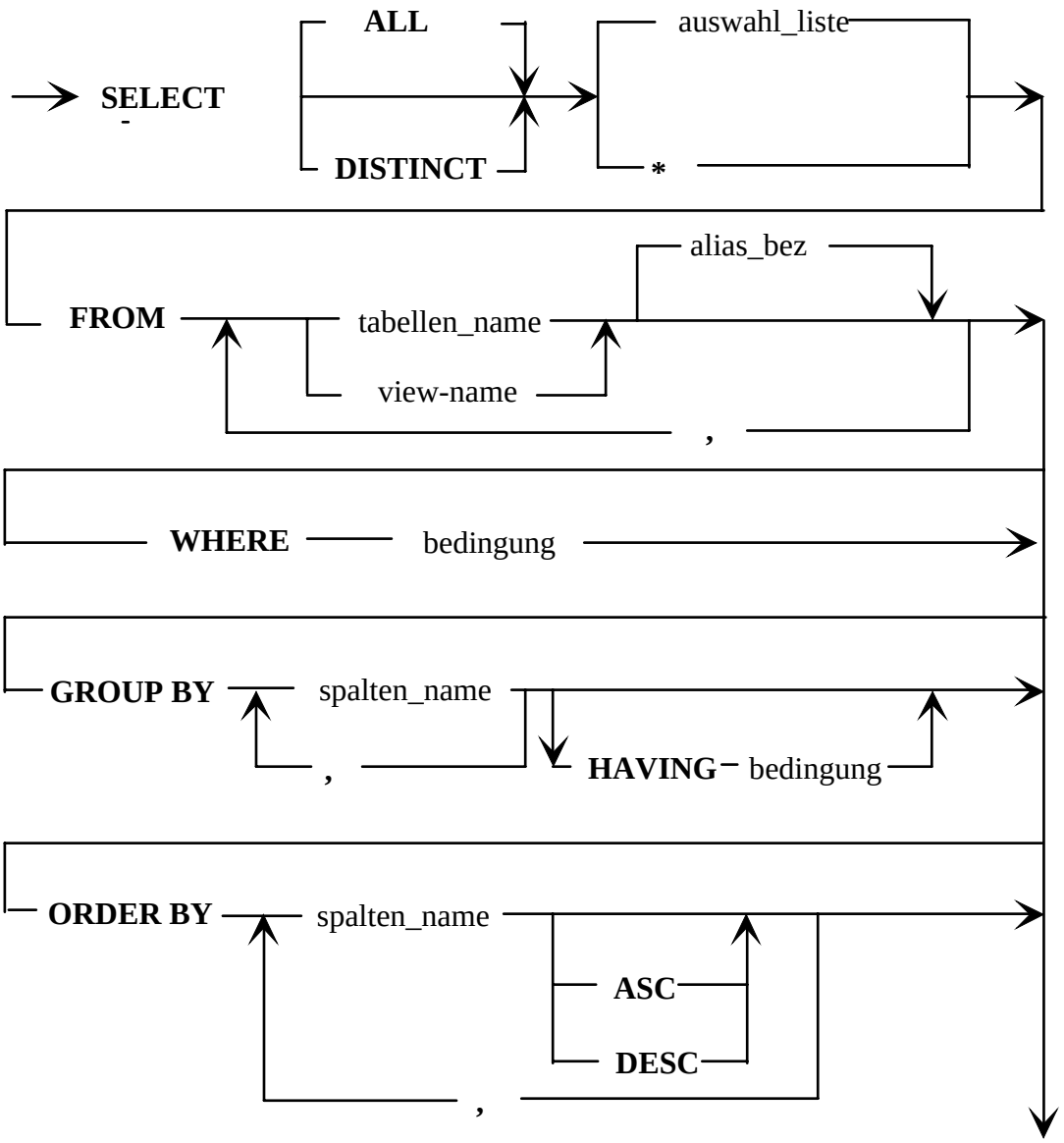
Der 1. Klausel legt fest, welche Spalten (Attribute) im Ergebnis (Abbildungsbereich) berücksichtigt werden sollen. Die 2. Klausel bezeichnet die Tabellen und Views (Sichten), aus denen Zeilen und Spalten abgefragt werden sollen. Die 3. Klausel ist optional und bestimmt, daß nur diejenigen Zeilen im Ergebnis angezeigt werden sollen, die einer bestimmten Bedingung genügen.
Der formale Aufbau des `select`-Befehls ist an bestimmte Regeln gebunden, die die folgenden Syntax-Diagramme[64] für den Standard von 1989 beschreiben:

---

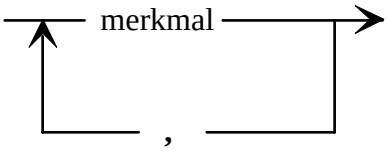
[62] vgl. Sandberg, G.: A primer of relational data base concepts, IBM Systems Journal, Heft 1, S.23 - 61, (1980)

[63] vgl. Zloof, M.M.: Query by Example: a database language, IBM Systems Journal, Heft 4, S. 324 - 342, (1977)

[64] Alle Bezeichnungen in nicht fettgedruckten Kleinbuchstaben werden entweder in anderen Diagrammen erläutert bzw. im Text beschrieben. Großgeschriebene Wörter in Fettdruck bezeichnen Schlüsselwörter bzw. terminale Symbole der select-Grammatik. Sie können unmittelbar genutzt werden. Wörter, die nur aus kleingeschriebenen Buchstaben bestehen, müssen ersetzt werden.

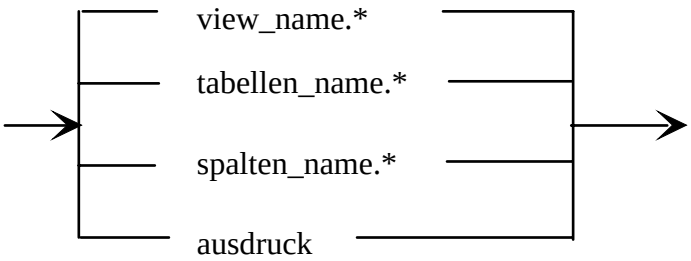


auswahl_liste

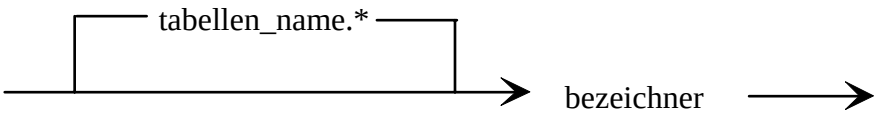


Die hier angegebene Grammatik des select-Befehls bezieht sich auf ENTRY-SQL von SQL-92. Ein nach diesen Vorschriften aufgebauter SQL-Befehl müßte von allen herkömmlichen, auf dem Markt befindlichen relationalen Datenbanksystemen verstanden werden.

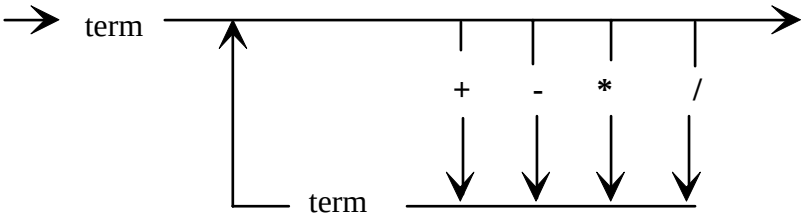
merkmal



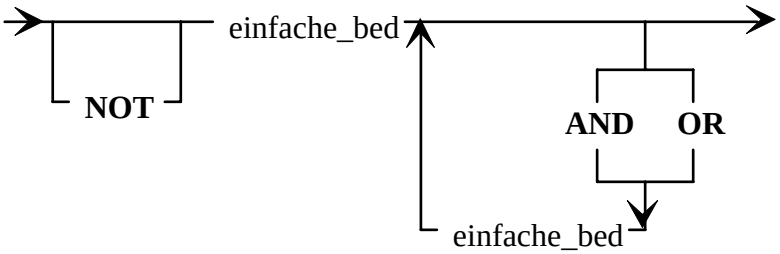
spalten_name



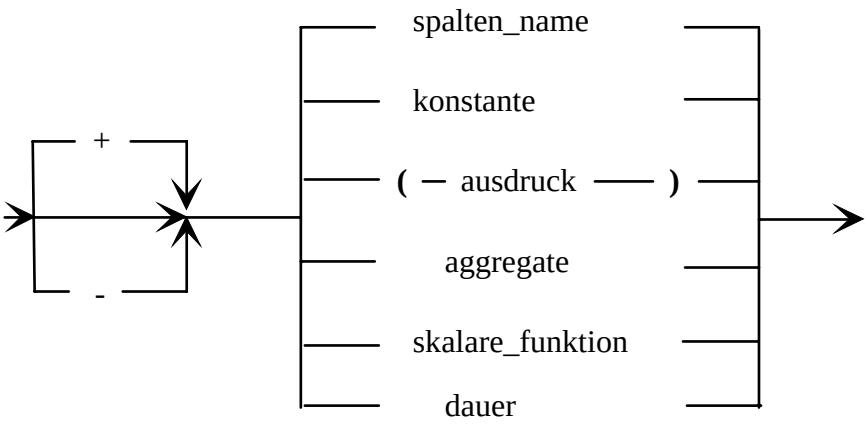
ausdruck



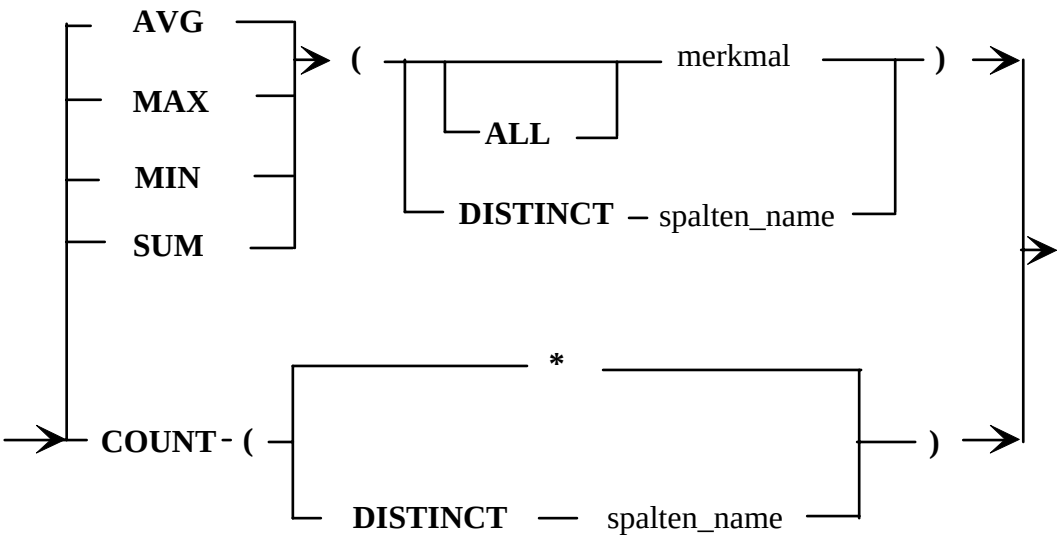
bedingung



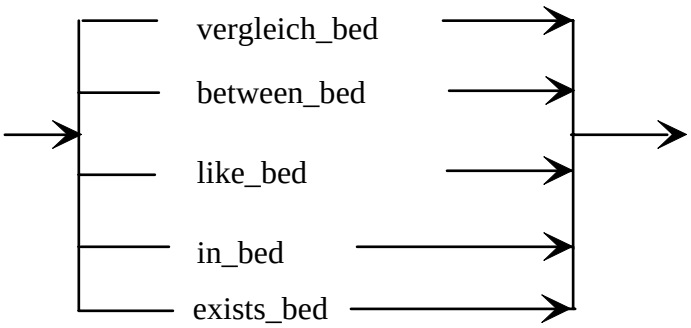
term



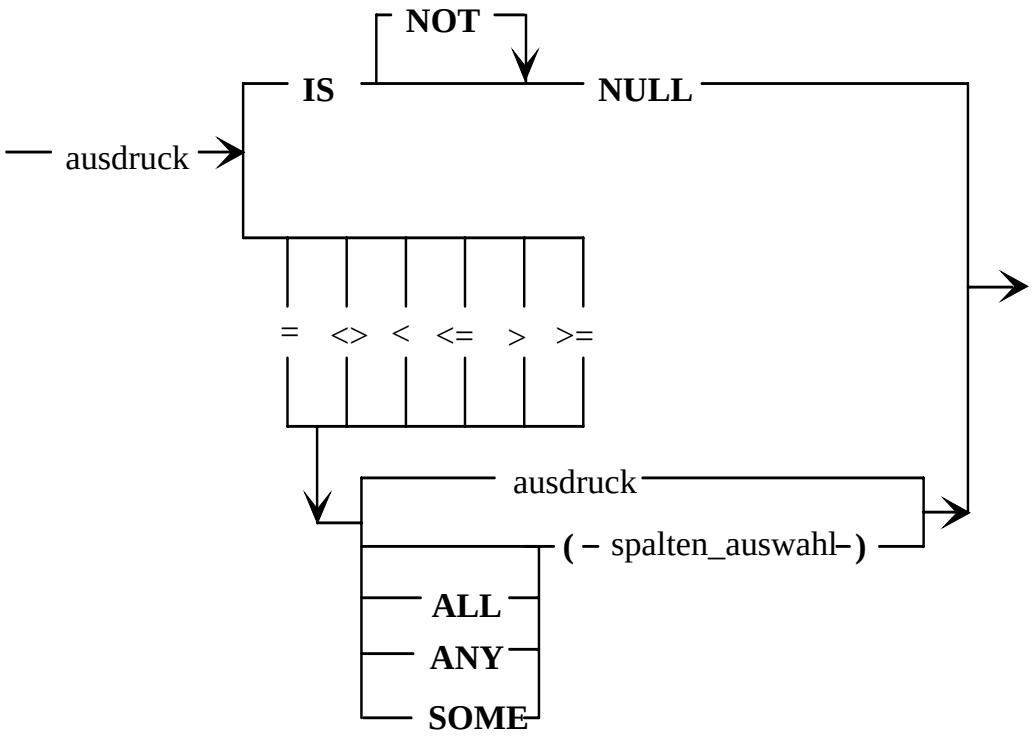
aggregate



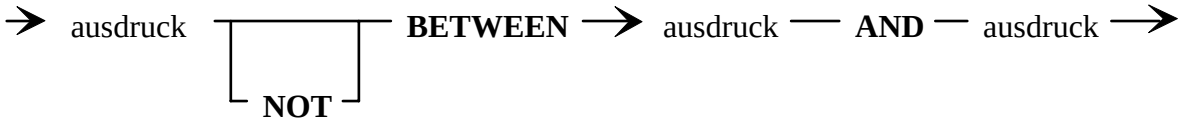
einfache_bed



vergleich_bed



between_bed



Der Operator BETWEEN vergleicht die angegebene Spalte mit einem Wertebereich einschl. der unteren und oberen Grenzwerte.

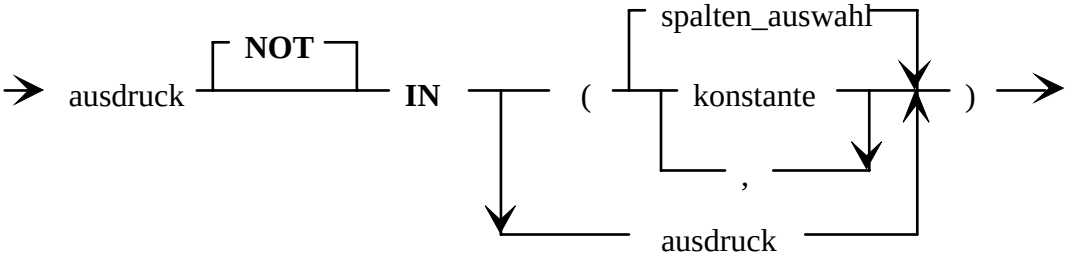
like_bed



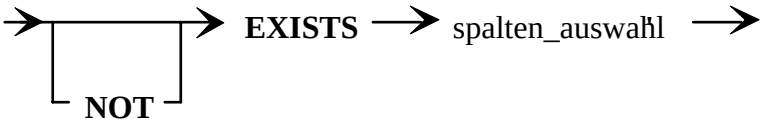
Der LIKE-operator verwendet sie als Textzeichenfolge angegebene Spalte mit einem Textmuster (string).
Bei "string" in like_bed" haben "_" und "%" eine spezielle Bedeutung:

- "_" steht für irgendein Symbol, d.h. der Vergleich geht über genau ein Zeichen mit dessen Stelle im Muster
- "%" steht für irgendeine Folge von 0 oder mehr Zeichen, d.h. der Vergleich geht über keines oder mehr Zeichen im Muster

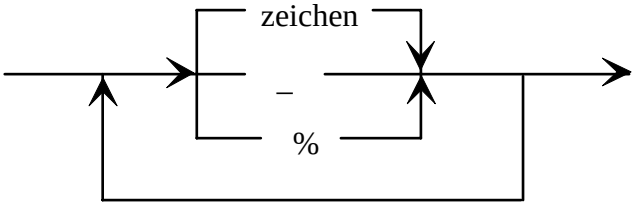
is_bed



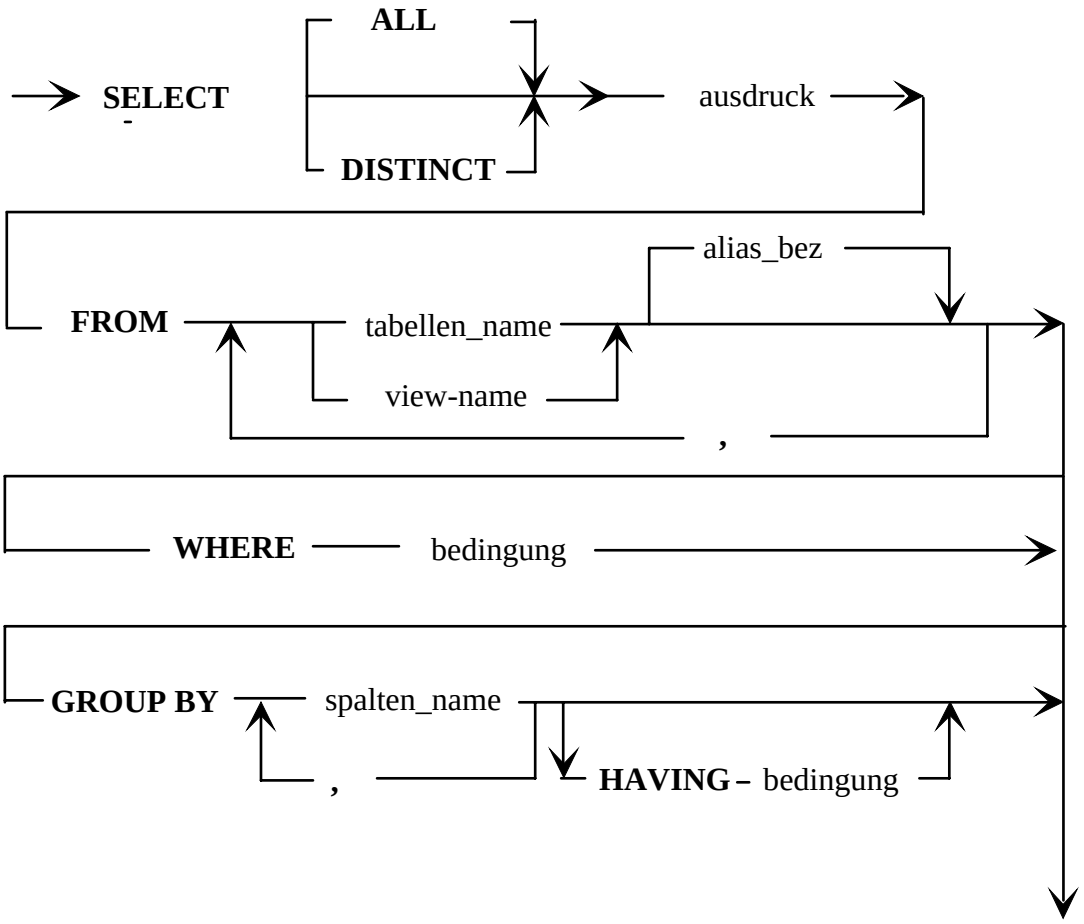
exists_bed



string



spalten_auswahl



union_query

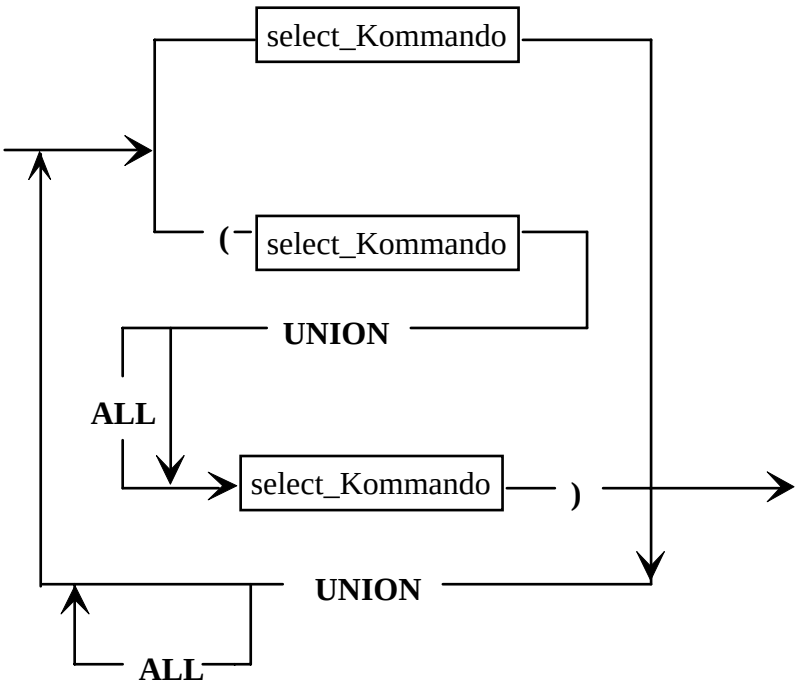


Abb. 1.4-2: Syntaxdiagramme zum SQL-Kommando

- Die Basis der „select“-Anweisung

Jeder SQL-Befehl bzw. jede SQL-Anweisung beginnt mit einem Befehlswort, das die auszuführende Operation bezeichnet, z.B. **select**. Zu vielen SQL-Befehlen gehören eine oder mehrere Klauseln, mit denen der Befehl für eine spezielle Anforderung angepaßt wird. Jeder SQL-Befehl muß zwei Bedingungen erfüllen:

1) Bezeichnen der gewünschten Daten
   (eine Menge von Zeilen in einer oder mehreren Tabellen)
2) Bestimmen, was mit den Daten geschehen soll.

Die Anweisung **select-from-where** gibt dem Benutzer immer eine Resultatstabelle zurück.
Tabellen werden über die sogenannte „leere Bedingung“ aufgelistet, z.B. die Tabelle „`angestellte`“[65]:

```
SELECT *
FROM ANGESTELLTE;
```
[66]

Man kann sich auch bei der Auswahl der Ausgabespalten auf bestimmte Felder beschränken:

```
SELECT ANG_ID, NAME, GEBDAT
FROM ANGESTELLTE;
```

Die ausgewählten Datensätze können über die zusätzliche (optionale) `ORDER BY` - Klausel in eine sortierte Reihenfolge gebracht werden:

```
SELECT *
FROM ANGESTELLTE
ORDER BY NAME;
```

Die Sortierreihenfolge kann auf- (`ASC`) oder absteigend (`DESC`) sein. Auch sekundäre bzw. tertiäre Sortierkriterien können gebildet werden:

```
SELECT *
FROM ANGESTELLTE
ORDER BY ANG_ID ASC, ABT_ID DESC;
```

Wiederholungen von Tabellenspalten-Werten in der Ausgabe werden durch Angabe von `DISTINCT` vor dem Feldnamen ausgeschlossen.
Eine andere konkrete Anfrage an die vorliegende Tabelle könnte sein:

```
SELECT * FROM ANGESTELLTE
WHERE NAME = 'Fritz';
```

SQL kennt 6 relationale Operatoren zur Angabe von Bedingungen: „ `=, <>` `oder !=, <, >, <=, >=`“

Bsp.: Welcher Angestellte ist vor dem 2. Dezember 1957 geboren?

```
SELECT * FROM ANGESTELLTE
WHERE GEBDATUM < '02-DEC-57'
ORDER BY NAME;
```

---

[65] Vgl. 1.3.3
[66] SQL ist nicht `case sensitive`

- Komplexe Bedingungen

Mehrere Suchbedingungen können in einer WHERE-Klausel kombiniert werden, z.B.

```
SELECT * FROM ANGESTELLTE
WHERE NAME = ‘Fritz‘ AND JOB_ID = ‘SY‘;
```

Drei logische Operatoren **NOT**, **AND**, **OR** ermöglichen die logische Kombination von Suchbedingungen. Die angegebene Reihenfolge entspricht der Priorität ihrer Verarbeitung. Mit Klammern kann die Abarbeitungsfolge beeinflußt werden.

- IN, BETWEEN und LIKE

SELECT kann auch in Zusammenhang mit den Operatoren **BETWEEN**, **LIKE** und **IN** Suchbedingungen über die WHERE-Klausel festlegen. Die 2 Operatoren BETWEEN, LIKE ersetzen relationale Operatoren und legen den Geltungsbereich der WHERE-Klausel fest
Der mit **BETWEEN** festgelegte Bereich ist inklusiv: Auch Zeilen, deren Feldwert einem der beiden Grenzwerte entspricht, werden in der Ergebnistabelle aufgenom-men, genauso wie alle dazwischenliegenden Werte.

```
SELECT * FROM ANGESTELLTE
WHERE GEBDATUM BETWEEN '01-JAN-55' AND '31-DEC-55';
```

Der Operator **IN** prüft, ob ein Feldwert mit einem Wert in der angegebenen Werteliste übereinstimmt:

```
SELECT ABT_ID, BEZEICHNUNG
FROM ABTEILUNG
WHERE ABT_ID IN ('KO', 'OD')
ORDER BY ABT_ID;
```

LIKE ermöglicht die Suche nach Zeichenfolgen. Dabei ersetzt

- der Unterstrich (_) ein beliebiges einzelnes Zeichen
- das Prozentzeichen (%) eine beliebige Anzahl von Zeichen

Bsp.: Welche Angestellte haben einen Namen der mit 'F' beginnt?

```
SELECT * FROM ANGESTELLTE
WHERE NAME LIKE 'F%';
```

- Die Bedingung „IS NULL“

Zur Ermittlung von Tupeln, die NULL-Werte in einem Attribut besitzen, gibt es den speziellen Operator **IS NULL** bzw. IS NOT NULL. NULL-Werte sind nicht durch Zeichen repräsentiert, daher ist der Operator auch nicht gleichwertig mit einem "=". Bei einer Selektion durch eine ORDER BY-Klausel erscheinen NULL-Werte stets am Anfang (unabhängig von der gewählten Sortierfolge).

- Verbund

Es besteht grundsätzlich die Möglichkeit mehrere Tabellen im Rahmen von Verbundoperationen zu verknüpfen.
Bsp.:

1. Welche Angestellten üben den Beruf 'Systemplaner' aus?

```
SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME, JOB.TITEL
FROM ANGESTELLTE, JOB
WHERE ANGESTELLTE.JOB_ID = JOB.JOB_ID AND JOB.TITEL = 'Systemplaner';
```

2. Welchen Beruf übt welcher Angestellte aus?

```
SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME, JOB.TITEL
FROM ANGESTELLTE, JOB
WHERE ANGESTELLTE.JOB_ID = JOB.JOB_ID;
```

3. Bestimme alle Angestellten aufsteigend geordnet nach Berufen (job.titel) und innerhalb des Berufs absteigend nach Gehalt (job.gehalt). Die Ausgabe soll in eine Tabelle mit folgenden Spaltenüberschriften erfolgen:

```
      job.titel   job.gehalt    angestellte.name
```

```
SELECT JOB.TITEL, JOB.GEHALT, ANGESTELLTE.NAME
FROM JOB, ANGESTELLTE
WHERE JOB.JOB_ID = ANGESTELLTE.JOB_ID
ORDER BY JOB.TITEL, JOB.GEHALT DESC;
```

Über die SELECT-Anweisung kann eine Tabelle mit sich selbst verknüpft werden (Selbstverknüpfung, SELF JOIN). Mit Selbstverknüpfungen können Daten aus verschiedenen Zeilen einer einzigen Tabelle zueinander in Beziehung gesetzt werden, und diese Daten in einer Ergebnistabelle zu Zeilen zusammengefaßt werden. Der SELF JOIN kann über die Definition unterschiedlicher Namen für eine Tabelle realisiert werden. SQL behandelt die unter verschiedenen Namen geführten Tabelle wie unterschiedliche, eigenständige Tabellen.
Bsp.: Bestimme aus der Tabelle Angestellte, jeweils für einen Mitarbeiter die Angestellten, mit denen er in der gleichen Abteilung arbeitet. Die Ausgabe soll in eine Tabelle mit folgenden Spaltenüberschriften erfolgen:

A.ANG_ID A.NAME B.ANG_ID B.NAME A.ABT_ID

```
SELECT A.ANG_ID, A.NAME, B.ANG_ID, B.NAME, A.ABT_ID
FROM ANGESTELLTE A, ANGESTELLTE B
WHERE A.ABT_ID = B.ABT_ID AND
A.NAME <> B.NAME
ORDER BY A.NAME;
```

Für den Verbund einer Tabelle mit sich selbst wird genutzt, daß jeder Name einer Tabelle eine (gewissermaßen selektionsinterne) Abkürzung (alias) erhalten kann (durch Leerzeichen getrennt, dem Tabellennamen in der from-Klausel nachgestellt).

- Unterabfragen

Bedingungen in SELECT- (INSERT-, DELETE- oder UPDATE-) Kommandos sowie HAVING-Klauseln dürfen sogenannte "**Subqueries**" (**Unterabfragen**) enthalten. In diesem Fall wird ein Wert bzw. eine Menge von Werten in einer

Bedingung errechnet. Das Ergebnis dieser Berechnung wird direkt in die befreffende Anfrage eingesetzt, jedoch nicht gespeichert.
Eine Unterabfrage ist eine SELECT-Anweisung, die im Kontext einer anderen SQL-Anweisung verschachtelt wird. Die verschachtelte Unterabfrage wird untergeordnete Unterabfrage genannt. Der Teil, der die verschachtelte Unterabfrage enthält, heißt Haupt- oder übergeornete Abfrage.
Der Aufbau der übergeordneten "`select`"-Anweisung hängt ab von der Anzahl der Werte, die als Ergebnis der untergeordneten SELECT-Abfrage ausgegeben werden. Folgende Fälle sind möglich:

- Unterabfrage mit einem Ergebnis
- Unterabfrage mit mehreren Ergebnissen

Eine Unterabfrage mit einem Ergebnis ist:

```
SELECT JOB.TITEL, JOB.GEHALT
FROM JOB
WHERE JOB.JOB_ID =
 (SELECT ANGESTELLTE.JOB_ID FROM ANGESTELLTE
  WHERE ANGESTELLTE.NAME = 'Fritz');
```

Bsp.: Bestimme alle Angestellten, die diesselbe Funktion (Titel) wie ‘Fritz‘ haben oder in der gleichen Abteilung beschäftigt sind

```
SELECT NAME FROM ANGESTELLTE
WHERE ANGESTELLTE.JOB_ID =
 (SELECT ANGESTELLTE.JOB_ID FROM ANGESTELLTE, JOB
  WHERE  NAME = 'Fritz' AND ANGESTELLTE.JOB_ID = JOB.JOB_ID)
OR ABT_ID =
  (SELECT ABT_ID FROM ANGESTELLTE WHERE NAME = 'Fritz')
ORDER BY NAME;
```

Unterabfrage mit einem Ergebnis verwenden folgende Vergleichsoperatoren: =, <, >, >=, <=, <>

In Unterabfragen mit mehr als einem Ergebnis muß eine "`where`"-Klausel verwendet werden, die mehr als einen Wert akzeptiert. Eine solche Abfrage kann die Operatoren **IN**, **ANY**, **ALL** (mehrwertige Vergleichsoperatoren) verwenden, z.B.:

1) Welche Angestellten üben die Tätigkeiten eines Systemplaners bzw. eines Ingenieurs aus?

```
SELECT ANGESTELLTE.NAME
FROM ANGESTELLTE
WHERE ANGESTELLTE.JOB_ID IN
 (SELECT JOB.JOB_ID
  FROM JOB
  WHERE JOB.TITEL = 'Systemplaner' OR
        JOB.TITEL = 'Ingenieur');
```

bzw.

```
SELECT ANGESTELLTE.NAME
FROM ANGESTELLTE
WHERE ANGESTELLTE.JOB_ID = ANY
 (SELECT JOB.JOB_ID
  FROM JOB
  WHERE JOB.TITEL = 'Systemplaner' OR
```

```
        JOB.TITEL = 'Ingenieur');
```

2) Finde alle Mitarbeiter (Angestellte) in der Abteilung `'Konstruktion'`, die einen Beruf (`job.titel`) haben, der zu einem Mitarbeiter der Abteilung `'Organisation und Datenverarbeitung'` gleich ist.

```
SELECT ANGESTELLTE.NAME, JOB.TITEL
FROM ANGESTELLTE, JOB, ABTEILUNG
WHERE ABTEILUNG.BEZEICHNUNG = 'KONSTRUKTION' AND
      ABTEILUNG.ABT_ID = ANGESTELLTE.ABT_ID  AND
      ANGESTELLTE.JOB_ID = JOB.JOB_ID        AND
      JOB.TITEL IN
      (SELECT JOB.TITEL FROM JOB, ANGESTELLTE, ABTEILUNG
       WHERE ABTEILUNG.BEZEICHNUNG = 'Organisation und Datenverarbeitung'
             AND ANGESTELLTE.JOB_ID = JOB.JOB_ID);
```

3) Finde alle Mitarbeiter (Angestellte) in der Abteilung `'Konstruktion'`, die einen Beruf (`job.titel`) haben, der zu einem Mitarbeiter der Abteilung `'Organisation und Datenverarbeitung'` nicht gleich ist.

```
SELECT ANGESTELLTE.NAME, JOB.TITEL
FROM ANGESTELLTE, JOB, ABTEILUNG
WHERE ABTEILUNG.BEZEICHNUNG = 'Konstruktion' AND
      ABTEILUNG.ABT_ID = ANGESTELLTE.ABT_ID  AND
      ANGESTELLTE.JOB_ID = JOB.JOB_ID        AND
      JOB.TITEL NOT IN
      (SELECT JOB.TITEL FROM JOB, ANGESTELLTE, ABTEILUNG
       WHERE ABTEILUNG.BEZEICHNUNG = 'Organisation und Datenverarbeitung'
             AND ANGESTELLTE.JOB_ID = JOB.JOB_ID);
```

4) Finde alle Angestellten, die mehr verdienen als irgendein Mitarbeiter in der Personalabteilung. Die Ausgabe soll in eine Tabelle mit folgenden Spaltenüberschriften erfolgen:

```
 JOB.GEHALT  JOB.TITEL   ANGESTELLTE.NAME   ANGESTELLTE.ABT_ID
```

```
SELECT JOB.GEHALT, JOB.TITEL, ANGESTELLTE.NAME, ANGESTELLTE.ABT_ID
FROM ANGESTELLTE, JOB
WHERE JOB.GEHALT > ANY
(SELECT GEHALT FROM JOB, ANGESTELLTE, ABTEILUNG
 WHERE ABTEILUNG.BEZEICHNUNG = 'PERSONALABTEILUNG'
 AND ABTEILUNG.ABT_ID = ANGESTELLTE.ABT_ID
 AND ANGESTELLTE.JOB_ID = JOB.JOB_ID)
AND JOB.JOB_ID = ANGESTELLTE.JOB_ID
ORDER BY GEHALT DESC;
```

Die Vergleichs-Operatoren einer "`vergleich_bed`"[67] dürfen in Unterabfragen (Subquery) durch ALL, ANY oder SOME (Synonym für ANY) modifiziert werden. Eine Anfrage darf mehrere "Subqueries" enthalten, Unterabfragen dürfen geschachtelt sein. Es ist möglich, "IN" für “= ANY" und "NOT IN" für "!= ALL" zu setzen. Die "ORDER-BY"-Klausel ist in Unterabfragen nicht zugelassen.

- EXISTS

Unterabfragen können auch mit dem Operator **EXISTS** verknüpft werden. Der Ausdruck "`exists (select ......`)" führt zum logischen Wert "wahr", falls das Ergebnis des in Klammern eingeschlossenen "`select`-"Ausdrucks nicht leer ist.

[67] Vgl. Syntaxdiagramme Abb. 1.4-2

```
SELECT JOB.TITEL FROM JOB
WHERE EXISTS
 (SELECT * FROM ANGESTELLTE
 WHERE ANGESTELLTE.JOB_ID = JOB.JOB_ID)
ORDER BY JOB.TITEL;
```

Über **EXISTS** (auf Existenz prüfen) kann der Existenzquantor in SQL interpretiert werden. SQL hat den Allquantor nicht implementiert. Allerdings können Abfragen mit dem Allquantor immer mit Hilfe des negierten Existenzquantors formuliert werden. Es gibt komplexe Unterabfragen, die die Verwendung von EXISTS in negierter Form erforderlich machen
Bsp.: Wähle die Angestellten aus, die die Qualifikation 'Systemplaner' haben

```
SELECT * FROM ANGESTELLTE
WHERE NOT EXISTS
(SELECT * FROM JOB
 WHERE JOB.TITEL <> 'Systemplaner' AND
       JOB.JOB_ID = ANGESTELLTE.JOB_ID);
```

Die vorliegende Abfrage zeigt eine Anwendung mit dem Allquantor der Prädikatenlogik. Standard-SQL hat den Allquantor nicht implementiert. Abfragen mit dem Allquantor können mit Hilfe des negierten Existenzquantors formuliert werden, z.B.: „Wähle die Qualifikation so aus, daß keiner existiert, der nicht die Qualifikation ‘Systemplaner‘ (SY) besitzt“.
Mit EXISTS gibt es demnach die Möglichkeit, die Existenz einer Unterabfrage zu prüfen. Dabei wird als Wert einer leeren Unterabfrage nicht NULL, sondern stattdessen UNKNOWN erwartet.

- EXISTS prüft darauf, ob eine gebundene Abfrage überhaupt Werte enthält. Sollte die Unterabfrage leer sein, stimmt der Test. Anderenfalls wird die Bedingung als falsch bewerte.
- NOT Exists prüft dagegen genau auf das Gegenteil, nämlich darauf, daß die Unterabfrage keinen Wert enthält Ist dies der Fall, wird die Bedingung als wahr bewertet. Enthält die Unterabfrage dagegen Werte, so liefert der Vergleich ein falsches Ergebnis zurück.

Sollten keine Zeilen in der Unterabfrage gefunden werden, so wird die äußere Abfrage erst gar nicht ausgeführt und ergibt dann keinen Wert. Damit ist nicht das Ergebnis, sondern zunächst die Ausführung der äußeren Abfrage direkt von der Unterabfrage abhängig., sondern zunächst

- Aggregatfunktionen

Aggregat-Funktionen berechnen tupelübergreifend Summe, Durchschnitt, Aufzählungen. So zählt die Funktion `COUNT(*)` alle Tupel einer Tabelle (einschl. Duplikate und einschl. aller Tupel, die Nullwerte enthalten).

```
SELECT COUNT(*)
FROM ANGESTELLTE;
```

Das Symbol **(*)** bestimmt, daß sich die Funktion `COUNT()` auf die gesamte Zeile bezieht, nicht auf ein einziges Feld. SQL-Aggregat-Funktionen sind:

COUNT()
liefert die Anzahl der ausgewählten Zeilen
SUM()

liefert die Summe der Werte in einem numerischen Feld
MIN()
liefert den kleinsten Wert einer Zeichen- ,Datums- oder einer numerischen Spalte
MAX()
liefert den größten Wert einer Zeichen-, Datums- oder einer numerischen Spalte
AVG()
liefert den Mittelwert (Durchschnitt) der Werte eines numerischen Feldes (Spalte).

Aggregat-Funktionen können auch mit der WHERE-Klausel kombiniert werden, z.B.:

1) Stelle eine Liste von Angestellten zusammen, die am meisten verdienen

```
SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME
FROM ANGESTELLTE, JOB
WHERE ANGESTELLTE.JOB_ID = JOB.JOB_ID AND
 JOB.GEHALT = (SELECT MAX(JOB.GEHALT) FROM JOB);
```

2) Welcher Angestellte hat ein monatliches Einkommen, das über dem Durchschnittseinkommen liegt?

```
SELECT A.ANG_ID, A.NAME
FROM ANGESTELLTE A, JOB J
WHERE A.JOB_ID = J.JOB_ID AND
J.GEHALT > (SELECT AVG(JOB.GEHALT) FROM JOB);
```

Aggregat-Funktionen können mit der GROUP-BY-Klausel auf disjunkte Teilmengen einer Tupelmenge bezogen werden, z.B.:

1) Wieviel Angestellte sind in der Abteilung OD beschaeftigt?

```
SELECT ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG, COUNT(ANGESTELLTE.ANG_ID)
FROM ABTEILUNG, ANGESTELLTE
WHERE ABTEILUNG.ABT_ID = 'OD' AND ANGESTELLTE.ABT_ID = 'OD'
GROUP BY ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG;
```

2) Bestimme eine Liste, die die Identifikation, Bezeichnung einer Abteilung und die Zahl der Beschaeftigten in dieser Abteilung zeigt.

```
SELECT ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG, COUNT(ANGESTELLTE.ANG_ID)
FROM ABTEILUNG, ANGESTELLTE
WHERE ANGESTELLTE.ABT_ID = ABTEILUNG.ABT_ID
GROUP BY ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG;
```

- GROUP BY und HAVING

Die Anweisung **GROUP BY** ermöglicht es, Datensätze (Tupel) gruppenweise zu ordnen. Auf diese Gruppen lassen sich verschiedene Operationen (in der Regel mit Aggregatsfunktionen) ausführen. Die Klauseln GROUP BY und HAVING bewirken, daß sich diese Funktionen nicht mehr auf alle Zeilen einer Tabelle, sondern auf alle Zeilen innerhalb einer Gruppe auswirken.
Die **GROUP BY**-Klausel führt Zeilen aus der Ergebnistabelle einer SELECT-Anweisung zu Gruppen zusammen, in denen bestimmte Spalten den gleichen Wert haben. Jede Gruppe wird in der Ergebnistabelle zu einer einzigen Zeile reduziert.
Somit kann über **GROUP BY** projiziert werden auf

- Attribute, über die "gruppiert" wird
- Gruppen-Funktionen (, angewendet auf Attribute der Haupttabelle)
- Konstante

Mit der **HAVING**-Klausel kann der Anwender den einzelnen Gruppen bestimmte Bedingungen auferlegen. Sie ist eine Selektion von Gruppen. Da im Ausgangspunkt der HAVING-Klausel bereits Gruppen gebildet sind, kann nur nach Attributen dieser Zwischen-Ergebnistabelle selektiert werden. Mit Hilfe der HAVING-Klausel können Gruppen ausgewählt werden, die in die Ergebnistabelle aufgenommen werden sollen

```
SELECT ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG, COUNT(*)
FROM ABTEILUNG, ANGESTELLTE
WHERE ANGESTELLTE.ABT_ID = ABTEILUNG.ABT_ID
GROUP BY ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG
HAVING COUNT(*) > 1;
```

Die HAVING-Klausel wird erst nach der Gruppierung ausgewertet, denn sie schränkt nur die durch die Gruppierung erzeugten Datenmengen ein. Sie ist aber hier, obwohl eine WHERE-Klausel vor allen anderen Klauseln ausgeführt wird, unbedingt erforderlich, da

```
SELECT ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG, COUNT(*)
FROM ABTEILUNG, ANGESTELLTE
WHERE ANGESTELLTE.ABT_ID = ABTEILUNG.ABT_ID
 AND COUNT(*) > 1
GROUP BY ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG;
```

zu folgender Meldung führt: `GROUP FUNCTION NOT ALLOWED HERE`
Gruppen-Funktionen gehören niemals in ein Prädikat der `WHERE`-Klausel.

- UNION

Häufig möchte man Ergebnisse mehrerer Abfragen verknüpfen. Das kann mit Hilfe des Operators `UNION` geschehen.
Bsp.: Welche Job-Identifikationen gibt es in Angestellte oder Job?

```
SELECT JOB_ID
FROM ANGESTELLTE
UNION
SELECT JOB_ID
FROM JOB;
```

2. Erzeugen bzw. Löschen von Tabellen, Views (Sichten) und der Indexe

Tabellen werden über die Anweisung **`CREATE TABLE`** ... erzeugt, z.B.:

```
CREATE TABLE ABTEILUNG
 (ABT_ID CHAR(2) NOT NULL,
 BEZEICHNUNG CHAR(40));

CREATE TABLE JOB
 (JOB_ID CHAR(2) NOT NULL,
 TITEL CHAR(30),
 GEHALT DECIMAL(8,2));
```

```
CREATE TABLE ANGESTELLTE
 (ANG_ID CHAR(3) NOT NULL,
 NAME CHAR(10),
 GEBDATUM DATE,
 ABT_ID CHAR(2),
 JOB_ID CHAR(2));

CREATE TABLE QUALIFIKATION
(ANG_ID CHAR(3),
 JOB_ID CHAR(2));
```

Jede Tabellenspalte umfaßt Werte einer bestimmten Domäne. Domänen lassen sich über Angabe des Datentyps der Spaten und der Anzahl Zeichen definieren. Generell unterscheidet SQL vier Basistypen

Datentyp	Erläuterung
char(<Laenge X>)	Zur Speicherung von Zeichenketten der Länge X
integer	Zur Speicherung von ganzen Zahlen
decimal(<Laenge X, Dezimalstellen Y>)	Zur Speicherung von Zahlen im Festkommaformat. Das Festkommaformat kann insgesamt aus X Stellen und Y Stellen nach dem Dezimalpunkt bestehen
date	Die Datumsangaben sind in den Datenbanksystemen spezifisch formatiert

In `CREATE TABLE ..` ist nach dem Datentyp die Angabe `NULL` (Standardmäßiger Default-Wert) bzw. `NOT NULL` möglich. Damit wird festgelegt, ob eine Spalte `NULL`-Werte (d.h. keine Werte) enthalten darf oder nicht. Primärschlüssel sollten grundsätzlich mit der Option `NOT NULL` ausgestattet sein. `NULL`-Werte werden in allen alphanumerischen Datentypen durch Leer-Strings der Länge 0 repräsentiert.
Indexe können mit `CREATE INDEX ...` erstellt werden, z.B.:

```
CREATE UNIQUE INDEX ABT_IDX
ON ABTEILUNG (ABT_ID);
CREATE UNIQUE INDEX JOB_IDX
ON JOB (JOB_ID);
CREATE UNIQUE INDEX ANGEST_IDX
ON ANGESTELLTE (ANG_ID);
CREATE UNIQUE INDEX QUAL_IDX
ON QUALIFIKATION (ANG_ID,JOB_ID);
```

SQL verfügt über zwei Zugriffsmethoden: den sequentiellen und indexorientierten Zugriff. Beim sequentiellen Zugriff beginnt das System am Anfang der Tabellen zu suchen und arbeitet Satz für Satz durch die Tabelle, bis der gewünschte Datensatz gefunden ist. Bei umfangreichen Datenbeständen sollte man für jeden Satz einen Suchbegriff (Index) vereinbaren, der in einer Indextabelle abgespeichert wird. Der Zugriff auf die tatsächlich vorliegende Tabelle kann über die Indextabelle erfolgen. jedem Suchbegriff ist eine eindeutige Positionsangabe (= Satzzeiger) des Datensatzes auf dem externen Speicher zugeordnet.
Für Indexe gelten die folgenden Bearbeitungsregeln:

- Eine „`order by`“-Klausel in einer „`select`“-Anweisung kann jede Spalte einer Tabelle referenzieren – unabhängig davon, ob die Tabelle über einen auf dieser Spalte basierenden Index verfügt oder nicht.
- Ein Index kann sich nicht aus mehr als 16 Spalten zusammensetzen
- Ein Index speichert keine `NULL`-Werte
- SQL verfügt über keine Anweisung mit der der Inhalt eines Indexes überprüft werden kann.
- Ein Index kann nur für eine Tabelle, jedoch nicht für eine Datensicht (view) erstellt werden.

Tabellen bzw. Sichten (views) können auch durch Auswahl aus eine anderen Tabelle erstellt werden (`create table` ... bzw. `create view` ... mit einer Unterabfrage).

Bsp.: Erzeugen einer Tabelle „ABTKOSTEN“

```
CREATE TABLE ABTKOSTEN
(ABT_ID NOT NULL, BEZEICHNUNG NOT NULL,GEHAELTER)
AS SELECT DISTINCT ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG,
          SUM(JOB.GEHALT)
   FROM ABTEILUNG, ANGESTELLTE, JOB
   WHERE ABTEILUNG.ABT_ID = ANGESTELLTE.ABT_ID AND
         JOB.JOB_ID = ANGESTELLTE.JOB_ID
   GROUP BY ABTEILUNG.ABT_ID, ABTEILUNG.BEZEICHNUNG;
```

Eine Datensicht (**view**) ist eine (gespeicherte) Abfrage, deren Ergebnis auf einer Abfrage einer oder mehrerer Tabellen beruht und gespeichert ist.

Bsp.: Erstelle eine Sicht Einkommen, die Identifikation, Name, Abteilungsbezeichnung, Gehalt des Angestellten enthält

```
CREATE VIEW EINKOMMEN(ANG_ID, NAME, BEZEICHNUNG, GEHALT)
AS SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME,
     ABTEILUNG.BEZEICHNUNG, JOB.GEHALT
 FROM ANGESTELLTE, ABTEILUNG, JOB
 WHERE ANGESTELLTE.ABT_ID = ABTEILUNG.ABT_ID AND
       ANGESTELLTE.JOB_ID = JOB.JOB_ID;
```

```
SELECT * FROM EINKOMMEN;

ANG NAME       BEZEICHNUNG                                GEHALT
--- ---------- ---------------------------------------- ---------
A1  Fritz      Organisation und Datenverarbeitung            6000
A2  Tom        Konstruktion                                  6000
A3  Werner     Organisation und Datenverarbeitung            3000
A4  Gerd       Vertrieb                                      3000
A5  Emil       Personalabteilung                             3000
A7  Erna       Konstruktion                                  3000
A8  Rita       Konstruktion                                  3000
A9  Ute        Organisation und Datenverarbeitung            6000
A10 Willi      Konstruktion                                  6000
A12 Anton      Organisation und Datenverarbeitung            6000
A13 Josef      Konstruktion                                  6000
A14 Maria      Personalabteilung                             3000

12 rows selected.
```

Bsp.: Bestimme aus der vorliegenden Sicht "einkommen" mit Hilfe einer SQL-Anweisung eine Liste mit folgender Ausgabe:

- Identifikation, Name, Gehalt des Angestellten
- das durchschnittliche Gehalt, das in der Abteilung verdient wird, der der Angestellte angehört

```
SELECT A.ANG_ID, A.NAME, A.GEHALT, AVG(ALL B.GEHALT)
FROM EINKOMMEN A, EINKOMMEN B
WHERE A.BEZEICHNUNG = B.BEZEICHNUNG
GROUP BY A.ANG_ID, A.NAME, A.GEHALT;
```

Allgemein können über „views“ nicht nur Selektionen ausgeführt werden, sondern auch Einfügungen, Änderungen, Löschungen unter folgenden Bedingungen:

- Bie Löschungen über einen „view“ darf der „view“ keine GROUP BY- oder DISTINCT-Klausel enthalten, und das Pseudo-Attribut ROWNUM nicht verwenden. Anderenfalls könnten zu einem über den „view“ sichtbaren Datensatz in der Tabelle mehrere Datensätze existieren, die wegen der „DISTINCT“-Klausel nicht unterscheidbar wären
- Bei Änderungen über einen „view“ darf der Änderungswert nicht über einen Ausdruck bestimmt werden. Es gelten diesselben Bedingungen wie beim Löschen.
- Beim Einfügen über einen „view“ sind alle vorstehenden Restriktionen zu beachten, und es darf keine „NOT NULL“-Bedingung der realen Tabelle verletzt werden, d.h. alle „NOT NULL“-Felder müssen im „view“ enthalten sein.

Mit **`DROP TABLE ...`** / **`DROP VIEW ...`** / **`DROP INDEX ...`** können Tabellen, Sichten, Indexe gelöscht werden, z.B.:

```
DROP VIEW EINKOMMEN;
DROP TABLE ABTKOSTEN;

DROP INDEX ABT_IDX;
DROP INDEX JOB_IDX;
DROP INDEX ANGEST_IDX;
DROP INDEX QUAL_IDX;

DROP TABLE QUALIFIKATION;
DROP TABLE ANGESTELLTE;
DROP TABLE JOB;
DROP TABLE ABTEILUNG;
```

3. Ändern der Tabellenstruktur

Zum Einfügen neuer Felder dient der `ALTER TABLE` ... –Befehl mit der `ADD`-Klausel, z.B.:

```
ALTER TABLE ABTKOSTEN ADD(ANZAHL INTEGER);
```

Reicht der zur Verfügung gestellte Platz von Tabellenspalten nicht aus, dann kann über `ALTER TABLE` ... mit einer `MODIFY`-Klausel die Bereichsgröße dieser Felder erweitert werden.

4. Hinzufügen von Datensätzen (Zeilen)

Zum Einfügen eines einzelnen neuen Datensatzes dient der `INSERT`-Befehl mit einer `VALUE`-Klausel, z.B.:

```
insert into abteilung values
 ('KO','Konstruktion');
.....................
insert into job values
('KA','Kaufm. Angestellter',3000.00);
.....................................
insert into angestellte values
 ('A1','Fritz','02-JAN-50','OD','SY');
.....................................
insert into qualifikation values
('A1','SY');
.....................................
```

5. Aktualisieren von Daten

Beim Ändern von Sätzen werden die zu verändernden Felder angegeben und die zu ändernden Sätze durch eine Selektion bestimmt, z.B.:

```
update abtkosten
set anzahl =
 (select count(angestellte.abt_id)
 from abtkosten, angestellte
 where abtkosten.abt_id = 'KO' and
       angestellte.abt_id = 'KO'
 group by abtkosten.abt_id)
where abtkosten.abt_id = 'KO';
```

Falls die WHERE-Klausel fehlt, werden alle Spalten nach der Angabe in der SET-Anweisung verändert.

6. Löschen von Daten

Das Löschen von Datensätzen kann durch eine Selektion spezifiziert werden:

```
DELETE
FROM <Tabellen-Name>
WHERE <Bedingung>
```

Weitere Klauseln des SELECT-Befehls sind in der DELETE-Anweisung nicht sinnvoll und daher auch nicht zugelassen. Ohne WHERE-Klausel wird der Inhalt det Tabelle gelöscht, z.B.:

```
DELETE * FROM ANGESTELLTE;
```

7. Zusammenfassung

```
ALTER TABLE <TABLE NAME> ADD|DROP|MODIFY (<Spalten
Spezifikationen...>[68]);
```

```
COMMIT;
```

Macht alle Veränderungen seit der letzten Transaktion permanent und schließt die aktuelle Transaktion ab. Die mit `INSERT`, `DELETE` und `UPDATE` gesetzten Änderungen werden nicht sofort der Datenbank übergeben. Der Benutzer sieht zwar eine konsistente Sicht seiner Änderungen, andere Benutzer sehen diese Änderungen aber noch nicht. Erst wenn der Befehl `COMMIT` gegeben wird, gehen die gesammelten Änderungen in die Datenbank ein und werden öffentlich.

```
CREATE [UNIQUE] INDEX <Indextabellen-Name>
ON <TABLE NAME> (<SPALTEN LISTE>);[69]
```

```
CREATE TABLE <Tabellen-Name>
(<Spalten-Name> <Datentyp> [(<Größenangabe>[70])]
<Spaltenbedingung>[71],...<weitere Spalten>);[72]
```

[68] Vgl. CREATE-Befehl

[69] UNIQUE steht in eckigen Klammern und ist deshalb eine optionale Angabe

[70] Die Größenangabe ist nur bei bestimmten Datentypen erforderlich

[71] NULL bzw. NOT NULL und UNIQUE.

```
CREATE TABLE <Tabellen-Name>
[(Spaltendefinition1, ... , Spalten-definitionN[73])]
AS
<select-Anweisung>[74];

CREATE VIEW <Tabellen-Name> AS <Abfrage>;

DELETE FROM <Tabellen-Name> [WHERE <Bedingung>];

INSERT INTO <Tabellen-Name> [(<Spalten-Liste>)]
VALUES (<Wert-Liste>);

ROLLBACK;
```

Macht alle Veränderungen an der Datenbank seit dem letzten `COMMIT`-Kommando wieder rückgängig.

```
SELECT [DISTINCT|ALL] <Spalten-Liste, Funktionen,
                       Konstanten, etc.>
FROM <Tabellennamen-Liste oder Namen von Views>
[WHERE <Bedingung(en)>]
[GROUP BY <Gruppierte Spalte(n)>]
[HAVING <Bedingung>]
[ORDER BY <Geordnet nach Spaltenwerte> [ASC|DESC]];

UPDATE <Tabellen-Name>
SET <Spalten-Name> = <Wert>
[WHERE <Bedingung>];
```

`SQL` arbeitet mit verschiedenen Objekten (Tabelle, Spalte, Zeile, View, skalare Werte) und Operationen auf diesen Objekten, wie die verschiedenen Syntax-Diagramme zum `SELECT`-Kommando zeigen. `SQL` besteht aber nicht nur aus der `SELECT`-Anweisung, sondern setzt sich insgesamt aus 12 Befehlen zusammen. Diese Befehle lassen sich zur Datendefinitions-Sprache (Data Definition Language, **DDL**), der Datenmanipulationssprache (Data Manipulation Language, **DML**) und der Datenkontroll-Spache (Data Control Language, **DCL**) zuordnen. Außerdem gehört zu SQL der **Systemkatalog** zur Verwaltung der Tabellenstruktur und Werkzeuge für "Performance"-Optimierung zur Beschleunigung der Arbeitsabläufe.
Die **DDL** gibt die Definition von Tabellen-, View- und Indexdefinitionen an und benutzt dazu die Befehle: CREATE, DROP, ALTER
Die **DML** vollzieht das Suchen, Einfügen, Aktualisieren und Löschen und benutzt dazu die Befehle: `SELECT`, `INSERT`, `UPDATE` und `DELETE`.
Die **DCL** umfaßt drei verschiedene Arbeitsgebiete:

-Recovery und Concurrency (Transaktionen und Regeln für die Verfahrensweise bei Mehrfachzugriffen) mit den Befehlen `COMMIT`, `ROLLBACK`, `LOCK`
- Sicherheit (bzgl. der Zugriffsrechte) mit den Befehlen `GRANT`, `REVOKE`
- Integrität (Einschränkungen für den Erhalt der Korrektheit von Daten)

---

[72] Die Angaben zu den Spalten sind auch in Verbindung mit ALTER TABLE ... möglich.
[73] Namen der Spalten, die mit den von der Unterabfrage zurückgegebenen Werten assoziiert werden sollen
[74] Zulässige „select“-Anweisung, die beim Erzeugen der neuen Tabelle verwendet wird.

In gewisser Hinsicht ist SQL gegenwärtig die standardisierte Datenbanksprache. Es bestehen jedoch (- wie in fast jedem Standard -) Erweiterungen der jeweiligen SQL-Hersteller, die eine vollständige Kompatibilität nicht möglich machen. Alle sinnvollen Erweiterungen werden jedoch von den Normungsgremien ISO[75] und ANSI für eine umfangreiche Sprachdefinition gesammelt. 1992 wurde eine wesentlich erweiterte Fassung der SQL-Norm (**SQL2**) [76] veröffentlicht. Sei längerer Zeit wird bei ISO (parallel zu SQL2) am Projekt SQL3 gearbeitet und folgende Erweiterungen zu SQL2 diskutiert:

- Unterstützung komplexer Datenstrukturen
- Sprachmittel (Ausdrucksmöglichkeiten) für Datenbankprozeduren
- objektorientierte Konzepte, z. B. abstrakte Datentypen
- Unterstützung verteilter Datenbanken

Realisiert wurden im Rahmen von **SQL-99** folgende Erweiterungen:

- Neue Basistypen: BOOLEAN, BLOB (Binary Large Object), CLOB (Character Large Object)
- Neue Typkonstruktoren: ROW, ARRAY, REF
- Benutzerdefinierte Datentypen (Distinct Typ und strukturierter Typ)
- Typhierarchien (Subtypen)
- Typisierte Tabellen und Tabellenhierarchien (Subtabellen)
- Typisierte Sichten und Sichthierarchien

SQL:2003, die neueste Version der SQL-Norm, sieht die Unterstützung von XML-Daten vor. Es wird möglich, Tabellen mit XML wertigen Spalten anzulegen

SQL ist Bestandteil zahlreicher Datenbanksysteme (z.B. dBASE IV, Oracle, Database Manager[77] bzw. IBM Database 2 (DB 2).

1.5 Klassifikation der DB-Anwendungen

1.5.1 Elementare Anwendungsformen

Stapelverarbeitung

DB-Anwendungen im Stapelbetrieb werden heute noch dann priorisiert, wenn ein hoher Durchsatz der DB-Programme angestrebt wird

Dialogverarbeitung

Sie steht unter der Prämisse, mit möglichst geringer Antwortzeit und ohne Programmieraufwand auf jede **DB** zugreifen zu können. Eine Tendenz zum Ausbau der Online-Anwendung vom reinen Abfrage- und Erfassungsbetrieb zu Online-Update (Abfrage, Erfassen, Ändern, Löschen) oder gar Realzeitverarbeitung ist unverkennbar.

---

[75] ISO 9075: 1987: Database Language SQL 1987 bzw. ISO 9075: 1989: Database Language SQL with Integrity Enhancement 1989

[76] ISO 9075: 1992: Database Language SQL, 1992

[77] Es handelt sich hierbei um ein Datenbanksystem, das unter OS/2 läuft und wird im Rahmen von OS/2-Auslieferungen der IBM bereitgestellt

Interaktive Verarbeitung datenorientierter Aufgabenstellungen

Datenbanken lösen datenorientierte Aufgabenstellungen in interaktiver Verarbeitung. Ein derartiges Anwendungssystem kann in drei Basiskomponenten gegliedert werden:

- eine Präsentationskomponente zur Realisierung der Benutzerschnittstelle
- eine Logikkomponente zur Ausführung von Verarbeitungsfunktionen und Über-nahme des Kontrollflusses in Anwendungssystemen
- eine Datenkomponente mit der Aufgabe der Datenverwaltung

Die Datenkomponente kann weiter (sog. **5-Schichten-Architektur**)[78] unterteilt werden in:

- das Datenbanksystem
  Das ist heute in der Regel ein relationales Datenbanksystem. Anforderungen an das System werden in SQL formuliert. Das System löst mengenorientierte Anforderungen in satzorientierte Zugriffe (Sätze von Tabellen) auf. Im Rahmen der 5-Schichen-Architektur spricht man hier von der mengenorientierten Schnittstelle. Diese Schnittstelle umfaßt eine Klasse von Hochsprachen, z.B. SQL.
- das Datensatzverwaltungssystem (Recordmanagementsystem)
  Es löst Anforderungen nach Datensätzen über die Kenntnis von Zugriffspfaden (z.B. über einen Index) auf und bestimmt die Position der angeforderten Sätze in den Dateien (Tabellen). Das Datensatzverwaltungssystem setzt Aufrufe an das Dateisystem ab. Im Rahmen der 5-Schichten-Architektur spricht man von der satzorientierten Schnittstelle. Der Zugriff zur Datenbasis erfolgt satzweise.
- Pufferverwaltung, Einbringstrategie
  Von der satzorientierten Schnittstelle aus will man die Abspeicherung von Satzmengen steuern. Zur Kontrolle der Transportvorgänge von und zum Hinter-grundspeicher ist ein homogener und linearer Speicher wünschenswert, der Einzelheiten (z.B. Dateiorganisation, Pufferung, Blockgruppenanordnung) verdeckt. Homogen und linear bedeutet: Unterteilung in Adressierungseinheiten fester Größe (sog. Seiten), auf die direkt zugegriffen werden kann. Der Seitenorganisation wird häufig noch eine Segmentstruktur[79] überlagert, mit der sich die Zusammengehörigkeit der Daten ausdrücken läßt. Aufgabe dieser Schicht ist die Umsetzung der Segmente und Seiten auf Dateien und physische Blöcke (Einbringstrategie). Eine Pufferverwaltung sorgt dafür, daß benötige Seiten zur Verfügung stehen, ohne daß der Benutzer sich um Seitentransport oder Strategien zum Seitenaustausch kümmern muß.
- das Dateisystem
  Es stellt höheren Schichten Operationen zum Lesen, Einfügen, Ändern und Löschen von Bereichen in Dateien (Tabellen) zur Verfügung. Weitere Aufgaben sind: Das Sperren von Bereichen, die Manipulation von Dateien und Dateiverzeichnissen. Zur Realisierung dieser Aufgaben bedient sich das Dateisystem der tieferen Schicht eines Betriessystems.
- das System der physikalischen Ein-, Ausgabe
  Diese Komponente realisiert den Zugriff auf die Speicherperepherie. Diese Schicht greift über die Zugriffsmethode unmittelbar auf den Speicher zu und steuert die Übertragung von bzw. zum Systempuffer. Im Rahmen der 5-Schichten-Architektur spricht man hier von der Geräteschnittstelle (mit den diversen Treiberprogrammen).
  Damit werden Daten, die sich bspw. dem Benutzer im Relationenmodell noch als Tupel darstellen, vom Datenbankverwaltungssystem auf Seiten (d.s. Blöcke fester Länge in der Größenordnung von 512 Bytes bis 8 KBytes) abgebildet. Mengen derartiger Seiten sind jeweils zu linearen Adreßräumen zusammengefaßt, die Segmente[80] genannt werden. In den Seiten sind Records (Sätze) fester oder variabler Länge abgelegt. Solche Records können Tupel oder auch Verweislisten in einem Zugriffspfad sein. Recordoperationen werden demnach auf lesende und schreibende Seitenzugriffe abgebildet.

[78] vgl. 3.1.1

[79] Man spricht auch von der Segmentschnittstelle

[80] Synonyme: Realm, Area, DB Space

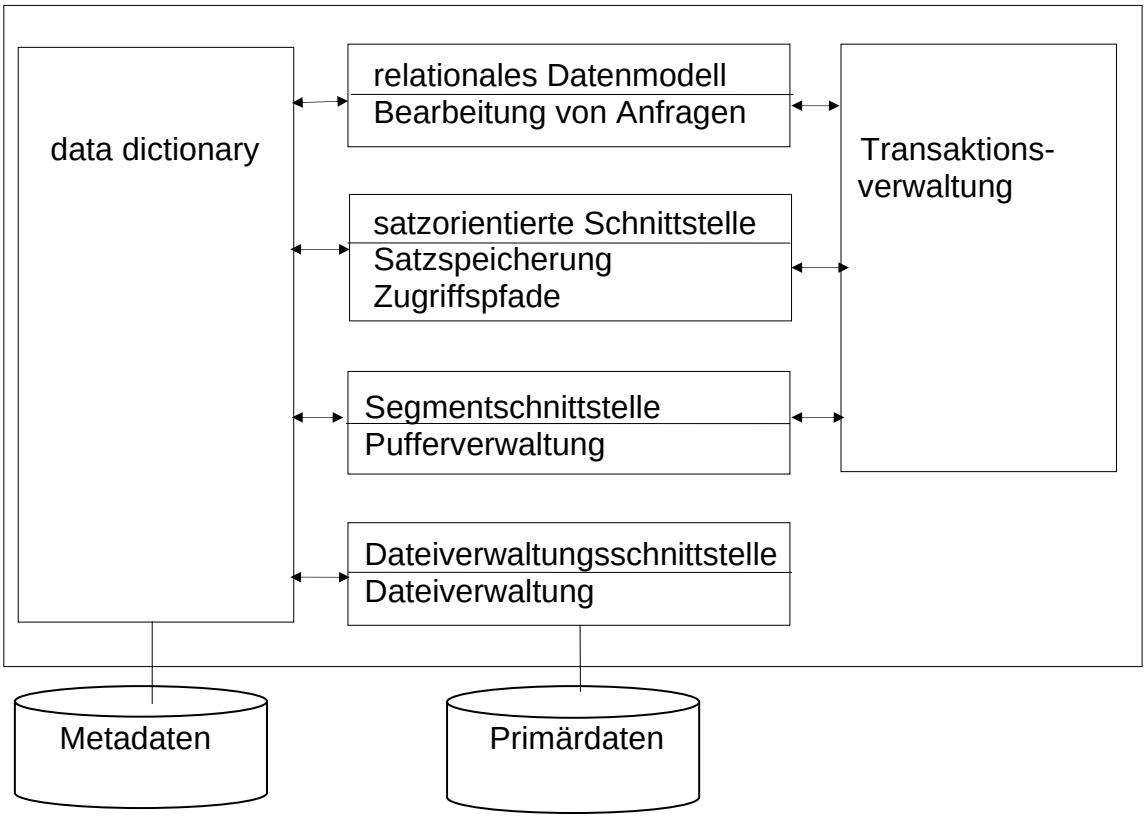


Abb. 1.5-1: Architektur von Datenbanksystemen

1.5.2 Transaktionsbetrieb

1.5.2.1 Transaktionen

Definition und Erklärungen

Bei größeren Datenbeständen ist es unumgänglich, daß sie von mehreren Benutzern gleichzeitig bearbeitet werden. So können häufig wiederkehrende Geschäftsvorgänge sofort durch Zugriff von Datenstationen über ein Anwendungsprogramm auf zentrale Datenbestände erledigt werden.
Eine (DC-) **Transaktion** (TA) ist als Folge von (Dialog-) Datenbank- und Verarbeitungsanweisungen definiert, die einen konsistenten Datenbestand in einen wieder konsistenten Bestand überführen. **Konsistenz** bedeutet: Der Datenbestand ist logisch widerspruchsfrei.
Eine Transaktion umfaßt demnach eine oder mehrere DB-Operationen, für die die DB-Software sog. Transaktionseigenschaften gewährleistet. Dazu zählt bspw. die "Alles-oder-nichts"-Eigenschaft: Änderungen einer Transaktion werden vollständig oder überhaupt nicht in die Datenbank übernommen. Erfolgreiche Transaktionen sind dauerhaft: Sie gehen trotz möglicher Fehlersituationen (Rechner-, Plattenausfall, Fehler im Kommunikationsnetzwerk) nicht mehr verloren. Dafür sorgen spezielle "Log-Dateien", die alle Änderungen protokollieren. Das DBMS sorgt bei Ausfällen für geeignete Recovery-Maßnahmen.

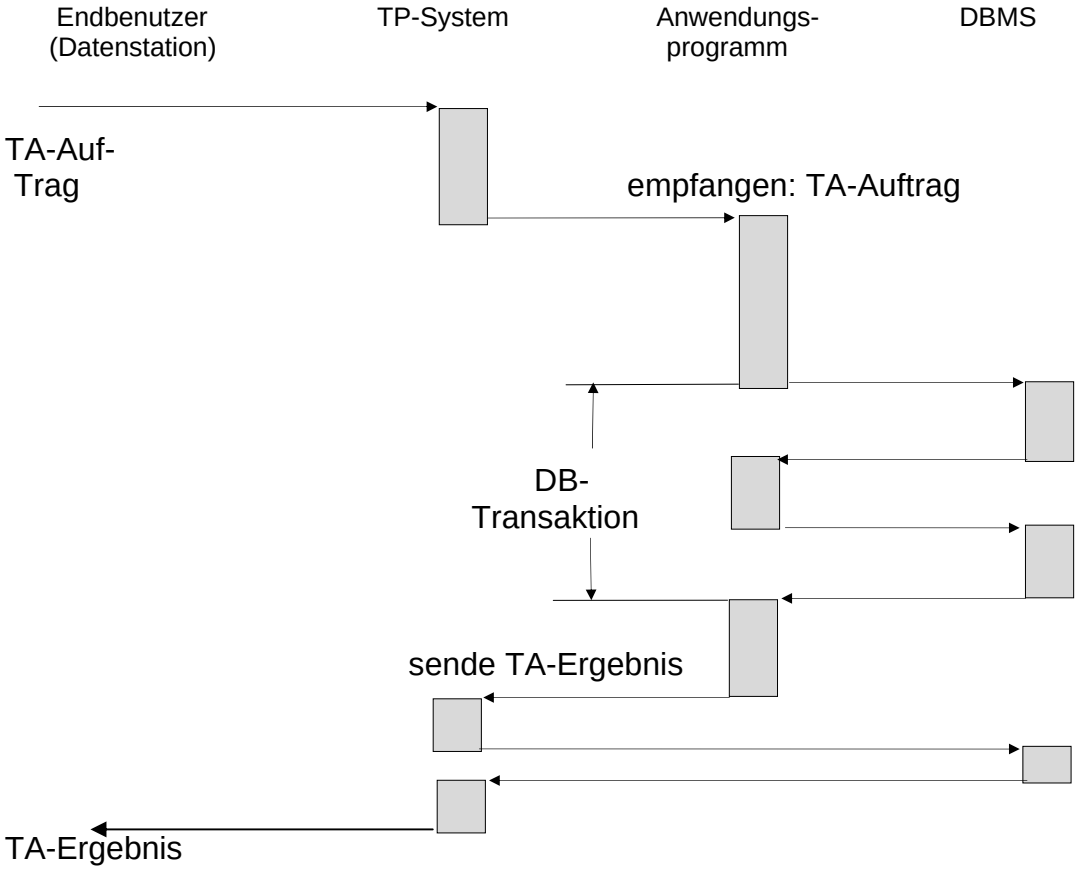


Abb. 1.5-2: Bearbeitung eines Transaktionsauftrages

Transaktionsparallelität

Im Mehrbenutzerbetrieb befinden sich in der Regel gleichzeitig mehrere Transaktionen in Bearbeitung. Von einer häufig sogar großen Anzahl von Endgeräten kommen Anforderungen (requests) , z.B. Auskunftsersuchen, Buchungsaufträge, Einzahlungen, Bestellungen, Rechnungen. Jedem Auftrag ist ein Anwendungsprogramm zugeordnet. Es nimmt Anforderungen vom Endgerät entgegen, führt die notwendigen Verarbeitungsschritte einschl. der Datenbankzugriffe aus und meldet das Ergebnis an das Endgerät zurück. Jede Instanziierung eines derartigen Anwenderprogramms wird als Transaktion ausgeführt. Das bedeutet: Die Ausführung einer Transaktion ist bzgl der Datenbank und bzgl. des Nachrichtensystems ununterbrechbar. Ein Synchronisationsmechanismus sorgt dafür, daß alle Programme sich so verhalten, als würden sie seriell (d.h. in Einbenutzer-Umgebung) ausgeführt.
Transaktionsparallelität entsteht durch „time sharing“. Die Anwendung bestimmt lediglich, welche und wieviel Arbeitsschritte zu einer Transaktion zusammengefaßt werden. Paralleltät zwischen Transaktionen steigert den Durchsatz durch Erhöhung der Zahl der Verarbeitungsprozesse. Ein Übermaß an „time sharing“ verlängert die Antwortzeit der einzelnen Transaktionen. Zu viele gleichzeitig aktive Transaktionen bewirken ein Übermaß an Synchronisierungskonflikten. Die Betriebssystemumgebung legt fest, wie viele Transaktionen bearbeitet werden sollen. Das DBS muß nur dafür sorgen, daß die **ACID**-Eigenschaft der Transaktionen (**A**tomicity, **C**onsistency, **I**solation, **D**urability) erhalten bleiben.

Die ACID-Eigenschaft

Transaktionen müssen die ACID Eigenschaften erfüllen:

- Atomarität (atomicity)
- Konsistenz (consistency)
- Isolierte Zurücksetzbarkeit (isolation)
- Dauerhaftigkeit (durability)

Unter **Atomarität** wird verstanden, dass alle zu einer Transaktion gehörenden Aktionen als eine logische Einheit angesehen werden und nur komplett ausgeführt werden dürfen ("Alles-oder-nichts"-Eigenschaft). Schlägt eine Aktion fehl, müssen alle bereits vorgenommenen Veränderungen rückgängig gemacht werden, die Transaktion wird zurückgesetzt.
Mit der **Konsistenz**-Eigenschaft wird gefordert, dass sich die Datenbank nach Ablauf der Transaktion in einen Zustand befindet, der die Konsistenz[81] bzw. Integrität[82] nicht verletzt. Kann dieser Zustand nicht erreicht werden, so muss zum Ausgangszustand zurückgekehrt werden.
Isolierte **Zurücksetzbarkeit** bedeutet, dass durch das Zurücksetzen einer Transaktion keine andere Transaktion so beeinflusst, dass diese ebenfalls zurückgesetzt werden muss. Die Transaktion muss alle Zugriffe auf gemeinsam genutzte Ressourcen serialisieren und garantieren, dass sich die konkurrierenden Programme nicht beeinflussen. Im Mehrbenutzerbetrieb mit vielen parallelen und

[81] Konsistenz: Der Datenbestand ist logisch wiederspruchsfrei.
[82] Integrität allg.: Übereinstimmung von realen und gespeicherten Daten.

überlappenden Transaktionen muss sich ein Programm also genauso verhalten wie im Einbenutzerbetrieb.

Unter **Dauerhaftigkeit**[83] versteht man, dass von einer erfolgreich abgeschlossenen Transaktion vorgenommenen Änderungen auch tatsächlich in der Datenbank wirksam werden und nicht mehr (z.B. in Folge eines Systemzusammenbruchs) einfach verloren gehen können.

Das Zustandsdiagramm einer Transaktion

Jeder im Mehrbenutzerbetrieb arbeitende Transaktion kann genau ein Zustand zugeordnet werden, in dem sie sich gerade befindet:

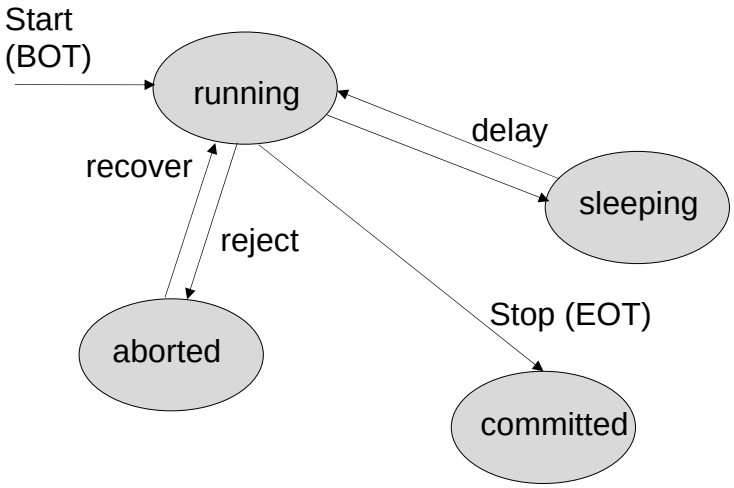


Abb. 1.5-3: Zustandsdiagramm einer Transaktion

Aktionen (logische Verarbeitungseinheiten) einer Transaktion sind durch BOT (Begin of Transaction) bzw. EOT (End of Transaction) begrenzt. Nach dem Start befindet sich eine Transaktion im Zustand "running". Hier wird ihr das Betriebsmittel Datenbank für eine bestimmte Zeit zur Verfügung gestellt. Ist der Auftrag danach abgewickelt, dann geht sie in den Zustand "committed" über, falls die EOT-Marke erreicht ist. Ist der Auftrag nicht vollständig abgewickelt, dann geht die Transaktion in den Zustand "sleeping" über. Die Bearbeitung wird zu einem späteren Zeitpunkt fortgesetzt. Kommt es während der Bearbeitung der Transaktion zu Konsistenzverletzungen, dann wird die Transaktion zurückgewiesen und geht in den Zustand "aborted" über. In diesem Fall wird zu einem späteren Zeitpunkt vollständig neu gestartet, alle Änderungen an der Datenbank werden rückgängig gemacht.

Fehler im Transaktionsbetrieb

Fehlerursachen können bspw. sein

a) Programmfehler
b) unerlaubt hoher Verbrauch von Ressourcen
c) Verlust von Hauptspeicherinhalten
d) Betriebssystemabsturz

[83] Persistenz ist ein Synonym für Dauerhaftigkeit

e) Zusammenbruch des Datenbanksystems
f) Plattenfehler (im schlimmsten Fall sog. Head-Crash)

Nur die Fehlertypen a) bis b) sind Transaktionsprogrammen unmittelbar anzulasten und können im laufenden Betrieb durch die „**Recovery**“-Komponente des Datenbanksystems repariert werden.
Fehler vom Typ c) bis e) heißen „Soft“-Crash. Maßnahmen zur Wiederherstellung der Datenbank nach solchen Fehlern werden unter dem Begriff „Crash Recovery“ zusammengefaßt. Sie erfordern einen Wiederanlauf des Datenbanksystems.
Nach einem Fehler vom Typ f) ist i. a. die aufwendige „Archiv-Recovery“ einzuleiten.

Anomalien im Mehrbenutzerbetrieb

Ein Reihe von Fehlern (Anomalien) kann im Mehrbenutzerbetrieb auftreten, selbst wenn jede einzelne Transaktion für sich allein feherfrei abläuft und das System reibungslos funktioniert. Derartige Anomalien können sein:

„lost update“
Die Ausführung unterschiedlicher Transaktionen auf eine gemeinsame Datenbank kann zu Problemen führen (z.B. „lost update“, „out-of-date retrieval“). So bewirken die folgenden Arbeitsschritte der Transaktionen T_1 und T_2, die nebeneinander (nebenläufig) ablaufen, den Verlust („lost update“) der durch die Transaktion T_1 durchgeführten Aktualisierung:

1. Transaktion T_1 holt das Objekt aus der Datenbank
2. Transaktion T_2 liest das Objekt aus der Datenbank
3. Transaktion T_1 aktualisiert das Objekt und schreibt es zurück
4. Transaktion T_2 aktualisiert das Objekt und schreibt es zurück in die Datenbank

„out-of-date retrieval“
Zu einer veralteten Wiedergewinnung („out-of-date retrieval“) führt dagegen die folgende Reihenfolge der Ereignisse:

1. Transaktion T_1 liest das Objekt zur Aktualisierung aus der Datenbank
2. Transaktion T_2 liest das Objekt zur Information
3. Transaktion T_1 verändert das Objekt und schreibt es zurück in die Datenbank

Lese-/Schreibsperren

Hier müssen für die Dauer der Ausführung einer Transaktion die betroffenen Daten (sog. Konsistenzbereich) gegen den Zugriff anderer Anwender geschützt werden. So muß eine der beiden Transaktionen den Zugriff bzgl. des Objekts sperren. Die Wahl bzw. die Definition des Konsistenzbereichs soll derart geschehen, daß nur der möglichst kleinste definierbare, aber notwendige Bereich gesperrt wird.
Eine **Lesesperre** erlaubt den Lesezugriff auf das Objekt und verhindert, daß eine Transaktion das Objekt aktualisiert. Eine beliebige Anzahl Transaktionen kann eine Lesesperre auf ein Objekt zur gleichen Zeit ausüben.

Ein **Schreibsperre** erlaubt Lese- und Schreibzugriff auf ein Objekt, hindert andere Transaktionen am Lesen und Schreiben des Objekts. Eine Transaktion erhält exklusiven Zugriff auf das Objekt.
Die Größe der Dateneinheiten, die in einem Datenbanksystem gesperrt werden können, wird als **Granularität** (degree of granularity) bezeichnet. In relationalen Datenbanksystemen sind typischerweise folgende sperrbare Einheiten bekannt:

- die Datenbank
- die Relation
- ein physischer Plattenblock
- ein Attributwert

Verklemmungen

Je nach Art der verwendeten Sperren entstehen zwischen Transaktionen Wartebeziehungen bzw. Serialisierungsbedingungen durch Zugriffe auf die Datenbank.
Falls Transaktionen Datenelemente sperren, kann es zu Verklemmungen kommen, z.B.: Die Transaktion T_1 und T_2 können nebenläufig ausgeführt werden. Das geschieht zufällig in folgender Reihenfolge:

1. Transaktion T_1 sperrt Objekt A zum Schreiben
2. Transaktion T_2 sperrt Objekt B zum Schreiben
3. Transaktion T_1 fordert eine Sperre auf Objekt B, muß aber warten, da Objekt B von T_2 gesperrt ist.
4. Transaktion T_2 fordert eine Sperre auf Objekt A, muß aber warten, da Objekt A von T_1 gesperrt ist.

An diesem Punkt können weder T_1 noch T_2 fortfahren. Sie sind verklemmt.

Synchronisation von Transaktionen (TA)

Es gehört zur Aufgabe der Transaktionsverwaltung, derartige unkorrekten Abläufe (Anomalien) voneinander zu isolieren und zu synchronisieren.
Die Abarbeitung einer Transaktion läßt sich durch das **Zwei-Phasen-Protokolls** ohne Gefährdung des Datenbestands und ohne Verklemmungen steuern und kontrollieren:

- Phase 1
Sperrphase des Konsistenzbereichs der Transaktion
Bevor eine Transaktion auf ein Objekt liest oder schreibt, muß sie das Objekt sperren
- Phase 2
Bearbeitung der Daten und Freigabe (auch sukzessiv) des Konsistenzbereichs
Hat eine Transaktion einmal eine Sperre wieder freigegeben, darf sie keine weiteren Sperren mehr anfordern bzw. erhalten, d.h.: In einer Transaktion sind alle Sperren vor allen Freigaben angeordnet.

Der Anwendungsprogrammierer soll sich aber nicht um derartige Maßnahmen zur Vermeidung von Anomalien kümmern müssen. Dieses Ziel kann erreicht werden, indem jede Transaktion automatisch so gesteuerrt wird, daß er scheinbar alleine auf der Datenbank arbeitet. Die sog. **Concurreny-Control**-Komponente der

Transaktionsverwaltung muß die Äquivalenz der parallelen Ausführung einer Menge von Transaktionen zu einer seriellen, d.h. nicht überlappenden Reihenfolge gewährleisten. Eine parallele Ausführung mehrerer Transaktionen mit dieser Eigenschaft heißt serialisierbar.

1.5.2.2 Das Zwei-Phasen-Commit-Protokoll

Dieses Protokoll wird verwendet um lokale Transaktionen an verschiedenen Programmen oder Maschinen (die Subtransaktionen) zu synchronisieren, so dass **entweder alle** erfolgreich durchgeführt werden **oder keines.**
Dabei nehmen die Subtransaktionen vor ihrem eigentlichen Commit einen Zwischenzustand, den ***Ready-to-Commit-Zustand***, ein. In diesem Zustand garantieren die Subtransaktionen, dass das Commit, falls vom Koordinator gewünscht, garantiert ausgeführt wird (selbst wenn zwischendurch ein Systemabsturz stattfindet) und dass aber auch noch ein Abbruch der Gesamttransaktion akzeptiert wird und dann die Subtransaktion zurückgesetzt wird, sog. rollback[84].
Ablauf des 2PC-Protokolls: Alle beteiligten Transaktionen werden durch eine „Prepare-to-Commit"-Anweisung des Commit-Managers aufgefordert, den Ready-to-Commit-Zustand einzunehmen. Die Subtransaktionen können wieder verteilte Transaktionen abgesetzt haben, zu welchen sie das Prepare-to-Commit Kommando weiterleiten müssen. So kann ein Transaktions-Baum entstehen mit dem Commit Manager als Wurzel.
Der Manager wartet dann auf die Bestätigung aller beteiligten Stationen, dass sie den Ready-to-Commit-Zustand eingenommen haben. Ist diese Bestätigung eingetroffen, wird sie in einem sicheren Platz zwischengespeichert, damit sich das System auch nach einem Wurzelknotenausfall in den Ready-to-Commit-Zustand zurückversetzen kann. Damit ist die erste Phase des 2PC Protokoll abgeschlossen.
Ein Commit wird vom Manager an alle Knoten gesendet. Haben die Subtransaktionen wiederum Subtransaktionen, so muss auch an diese die Commit- Aufforderung weitergegeben werden usw.. Auf diese Weise wird der gesamte Transaktionsbaum durchlaufen.
Die zweite Phase des 2PC- Protokolls ist beendet, wenn alle beteiligten Knoten eine Commit-Bestätigung zum Manager gesendet haben. Jetzt kann dem Client der erfolgreiche Abschluss der Transaktion mitgeteilt werden.
Die Transaktion wird abgebrochen, wenn eine Subtransaktion fehlschlägt und die Commit-Aufforderung zurückweist. Dann fordert der Manager alle Subtransaktionen (und diese wiederum ihre Subtransaktionen usw.) auf, ein Rollback durchzuführen. D.h. alle beteiligten Stationen machen die getroffenen Veränderungen rückgängig und versetzen sich wieder in den Ausgangszustand.

[84] Rollback: Zurücksetzen des Systems in den Ausgangszustand vor der Transaktion (z.B. werden Veränderungen der Daten in der DB rückgängig gemacht).

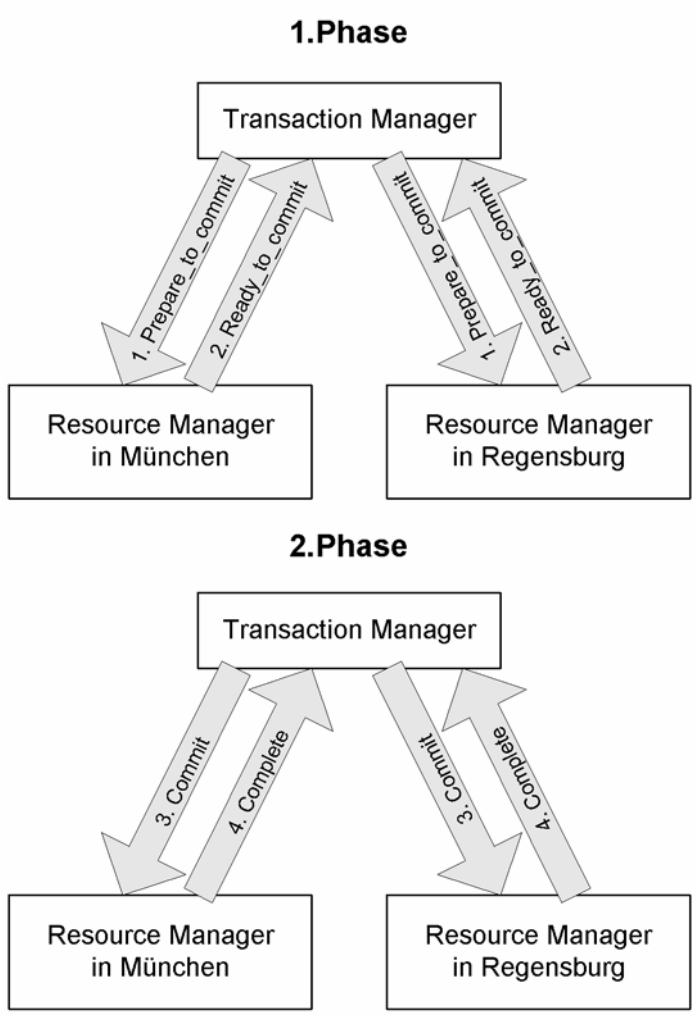


Abb.: Ablauf des 2PC-Protokoll

Der größte Nachteil des 2PC Protokoll ergibt sich aus der hohen Anzahl der Nachrichten die versendet werden müssen. Dies führt zu Performanz-Einbußen. Nicht jede Transaktion braucht die Sicherheit eines 2PC Protokolls (z.B. read-only Transaktionen). Diese müssen erkannt und durch ein einfaches Protokoll abgearbeitet werden.

1.5.2.3 Transaktionsmonitor (Operating Systeme für Transaction Processing)

Den Transaktionsbetrieb mit Datenbankanwendungen realisieren **DC**-Komponenten. Man spricht auch von einem „**Transaktionssystem** (TP-System, transaction processing system, **Transaktionsmonitor**)“. Dieses Systemprogramm koordiniert für viele Endbenutzer die unterschiedlichen Transaktionsaufträge und unterstützt dabei die logisch zusammenhängenden Verarbeitungssequenzen (Transaktionen).
Bei der Implementierung eines Transaktionssytems muß berücksichtigt werden: Eine (meist) große Anzahl von Benutzern ruft gleichzeitig Funktionen mit Zugriffen auf die gemeinsamen Datenbestände ab. Die Koordinierung übernimmt ein **Transaktionsmonitor** (TP-Monitor), der den Anwendungsprogrammen alle Fragen der Mehrfachausführung und der gerätespezifischen Ein-/Ausgabe abnimmt. Transaktionsprogramme greifen natürlich auf die Datenbank zu. **TP-Monitor** und Datenbanksystem müssen daher gleichzeitig aktiv sein und koordiniert zusammen arbeiten. Man unterscheidet nichtintegrierte DB/DC-Systeme von integrierten DB/DC-Systemen.
Nichtintegrierte DB/DC-Systeme entstehen durch einfache Kopplung der unabhängigen DB- und DC-Teile, die je einen eigenen Steuerteil aufweisen (z.B. UDS/UTM [85]). Die Koordinierung von "recovery" und "restart" ist in diesen Fällen kompliziert.
Bei integrierte DB/DC-Systemen ist DC-Komponente Bestandteil des Datenbank-systems.
Aus der Sicht der Anwender ist ein TP-Monitor ein Programm, das eine Endlosschleife mit einer ACID-Transaktion als Schleifenkörper bearbeitet.

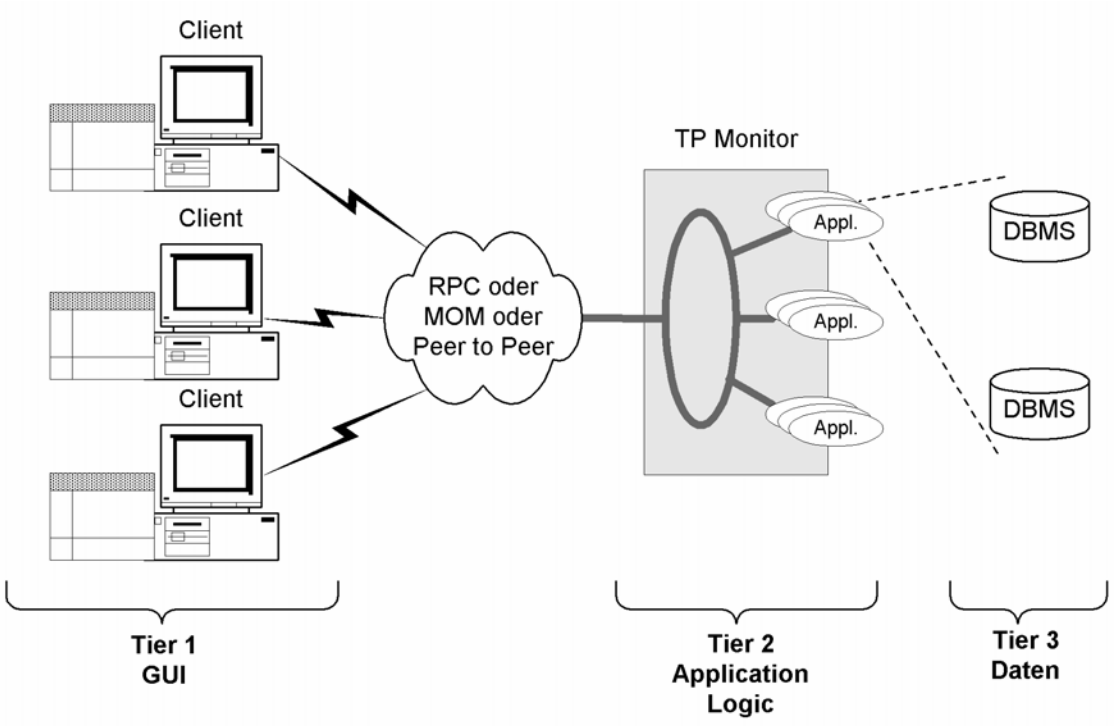


Abb.: Transaktionsmonitor, Beispiel für eine 3-Tier-Anwendung

[85] UDS ist ein "Universelles Datenbanksystem der Firma SIEMENS, UTM ist ein "TP"-Monitor der Firma SIEMENS

Der Kontrollfluss innerhalb des TP Monitor für jede Anfrage ist:

1. Den Input vom Display (oder einem anderen Gerät) in das Standard Format für Anfragen (XT API[86]) übersetzen.
2. Auswerten des Anfrage Headers um zu bestimmen, welche Transaktion gefordert wird und welches Programm die Anfrage abarbeiten soll.
3. Starten der Transaktion und des Programms welches in 2. bestimmt wurde (dieses Programm ruft typischerweise Aktionen auf einer Datenbank auf).
4. Commit zum Beenden der Transaktion im Erfolgsfall bzw. Abbruch im Fehlerfall.
5. Zurücksenden eines Output zum Display (oder einem anderen Gerät).

1.5.3 Client-Server-Systeme

1.5.3.1 Fernzugriff in Netzen aus autonomen Rechnern

Kommunikationsprotokolle

Es ist zweckmäßig, kleinere Datenverarbeitungssysteme zur Führung lokaler Datenbanken für Verarbeitungsaufgaben am Arbeitsplatz vorzusehen und diese über ein Netz an ein oder mehrere Rechenzentren anzuschliessen ("distributed processing"). Jeder Bildschirmarbeitsplatz hat einen eigenen Prozessor und Speicher. Das Anwendungsprogramm läuft auf diesem Arbeitsplatzrechner. Beim Fernzugriff auf die Datenbank findet eine Interprozeßkommunikation zwischen den Anwendungsprogrammen und dem Datenbankverwaltungssystem auf dem Zentralrechner statt.
Das **Client-Server-Modell** ist das dominierende Konzept für **verteilte Datenverarbeitung** mit über ein Kommunikationsnetz verbundenen Rechnern. Man versteht darunter eine Architektur, bei der eine Anwendung in einen benutzernahen Teil (Client, Frontend) und einen von allen Benutzern gemeinsam beanspruchten Teil (Server, Backend) aufgeteilt ist. Voraussetzung der verteilten Datenverarbeitung sind einheitliche Kommunikationsschnittstellen, ein Kommunikationssystem zur Unterstützung der Interprozeßkommunikation, Dienste für die Nutzung der Ressourcen, etc.
Die Kommunikation zwischen Rechnern erfordert Kenntnis und Einhaltung bestimmter Regeln. **Kommunikationsprotokolle** sind Regeln, nach denen Kommunikationspartner Verbindungen aufbauen, Information austauschen und wieder abbauen.
Von der International Standardisation Organisation (ISO) zusammen mit Open System Interconnection (OSI) wurde ein Architekturmodell entwickelt, das heute generell als Ausgangspunkt aller Kommunikationsprotokolle zwischen Rechnern verwendet wird. In diesem Modell werden die einzelnen für die Kommunikation zwischen Rechnern benötigte Dienste in 7 aufeinanderfolgenden Schichten abgewickelt. Jede Schicht erfüllt bestimmte Dienstleistungen bzw. Aufgaben und stellt das Ergebnis der nächsten Schicht zur Verfügung.

[86] XT API: X/Open Interface Standart für die Kommunikation zwischen Applikation und Transaktion Manager

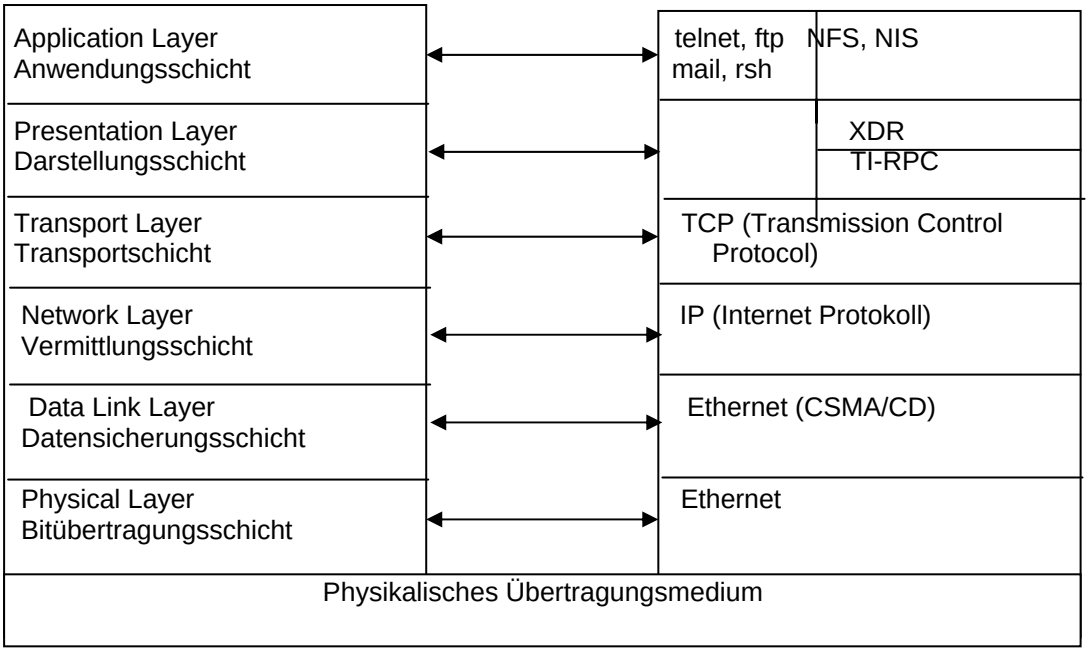


Abb. 1.5-4: OSI-Referenzmodell und ONC-Netzwerkprotokolle in einem UNIX-Betriebssystem

Ethernet beschreibt die Leistungen des LAN (Local Area Network) auf Bitübertragungs- und Datensicherungsschicht. Die Hauptaufgabe der Vermittlungsschicht (Network Layer) ist die Verkehrslenkung des Netzwerks. Nachrichten sollen über das Internet-Protokoll (IP) auf optimalem Weg ihr Zielsystem erreichen. Der Adressierungsmechanismus dieser Ebene basiert auf der Internet-Adresse. Die Transportschicht übernimmt den Transport von Nachrichten zwischen den Kommunikationspartnern, steuert den Datenfluß und definiert die Transportqualität (gerichtet / ungerichtet) der Datenübertragung. Das TCP (Transmission Control Protocol) beinhaltet verbindungsorientierte Dienste. Jede Ankunft eines TCP-Protokolls muß beantwortet werden.. Die Kommunikationssteuerschicht (Session Layer) steuert den Austausch von Nachrichten auf der Transportverbindung. Durch die Darstellungsschicht (Presentation Layer) wird die sog. Transfermatrix (Codierung für die zu übertragenden Daten) festgelegt. Die Anwendungsschicht stellt die Schnittstelle zum Anwendungsprozeß bereit. Protokollbeispiele für diese Schicht sind: Mail, FTP, Telnet, NFS, ...
Der langwierige ISO/OSI-Normungsprozeß hat dazu geführt, daß sich in weiten Bereichen andere Kommunikationsprotokolle durchgesetzen konnten, z.B.:

- IPX/SPX[87]
  IPX läuft auf der OSI-Schicht 3 (Vermittlung) und regelt den Transport von Daten zwischen den einzelnen Netzknoten sowie den Austausch von Statusinformationen. SPX ist auf der Schicht 4 (Transportschicht) angesiedelt und besorgt Auf- und Abbau von Verbindungen zwischen Client-PCs und Server-Anwendungen
- SNA[88]
- TCP/IP[89]
  Im Gegensatz zum OSI-Referenzmodell (und auch zu SNA) weist diese Kommunikationsschnittstelle nur 4 Schichten auf: Netzwerk, Internet, Transport und Anwendung.

[87] Protokollsammlung von Novell: Internet Package Exchange / Sequential Package Exchange
[88] Vernetzungskonzept von IBM: System Network Architecture
[89] Transmission Control Protocol / Internet Protocol

TCP/IP läßt sich auf verschiedenen Trägermedien (serielle Leitungen, X.25, Ethernet, usw.) einsetzen und ist vor allem im UNIX-Bereich weit verbreitet. Die Familie der TCP/IP-Protokolle hat sich bei Weitverkehrsnetzen (WAN) als Quasi-Standard durchgesetzt. Weltweit sind ungefähr zwei Millionen Rechner über das Internet[90] verbunden.

Die Unterstützung der verschiedenen Netzwerkmodelle (z.B. TCP/IP) und Betriebssysteme (z.B. Windows, Unix) erfolgt durch die einzelnen Software-Hersteller der DBS (Oracle, Sybase, Informix) bzw. durch die Werkzeug-Hersteller (z.B. Oracle, Microsoft), die Komponenten (z.B. ODBC-Treiber für besondere Plattformen und DBS) liefern.

Network-Operating-System

Das Network Operating System (NOS) hat die schwierige Aufgabe, den Service der im Netz erhältlich ist, als ein System darzustellen (single-system-image). Es setzt also die verteilten Teile des Netzwerkes zusammen und stellt sie als ein System dar - macht es also *transparent*.
NOS verwalten bspw. Datei- und Druckerressourcen und halten sie transparent, indem sie Anfragen über das LAN zu angeschlossenen Datei- und Druckerservern weiterleiten. Das NOS bietet dafür Proxies, die die Anfragen abfangen und an die entsprechenden Server weiterleiten.
Transparenz heißt also, dass das Netzwerk mit seinen Servern vor seinen Benutzern versteckt bzw. unsichtbar gemacht wird.

Kommunikation

Client/Server Applikationen sind über verschiedene Adressräume, physikalische Maschinen, Netzwerke und OS verteilt. Wie verläuft die Kommunikation?
Alle NOS bieten ein „peer-to-peer“ Interface, welches eine Kommunikation der Applikationen untereinander ermöglicht und sehr nah an der physikalischen Verbindung angesiedelt ist.
Die meisten NOS bieten auch eine Form des Remote Procedure Call (RPC), der die physikalische Verbindung versteckt und den Server nur “einen Funktionsaufruf entfernt“ erscheinen lässt.
Eine Alternative zu RPC bietet Message Oriented Middleware (MOM), die Messages zur Abarbeitung in eine Warteschlange stellt.

1. Peer-to-peer Kommunikation
Das Protokoll ist symmetrisch, jede Seite kann die Kommunikation iniziieren. Es maskiert nicht vollkommen die unterliegende physikalische Schicht. Das heißt, dass der Programmierer z.B. Übertragungstimeouts und Netzwerkfehler selbst abfangen muss.
Für eine einfache peer-to-peer Kommunikation wurde zuerst in Unix- Systemen und später auch in allen anderen OS **Sockets** eingeführt.
Sockets stellen logische “Steckdosen“ für die Herstellung von bidirektionalen Kommunikationsverbindungen bereit .
Der Socket-Kopf bildet die Schnittstelle zwischen dem Aufruf an das Betriebssystem zum Auf- und Abbau sowie Durchführung der Kommunikation und den weiter unten liegenden Systemschichten- dem Protokollteil und dem Gerätetreiber.
Die drei am meisten verwendeten Socket-Typen sind:

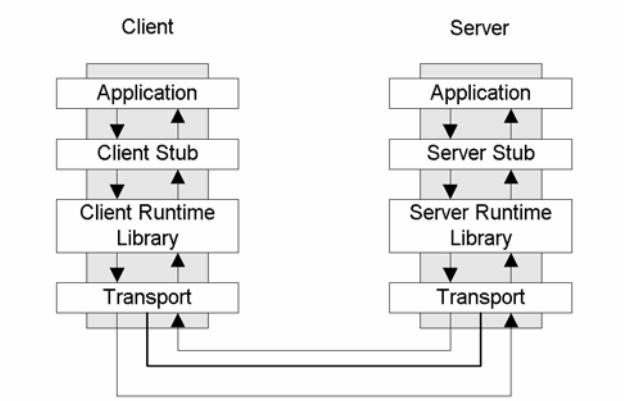
- Stream

[90] Weitverkehrsnetz, das etwa 4000 Netze mit über 6000 Rechnern verbindet

- Datagram
- Raw

Stream Sockets setzen auf das TCP, **Datagram Sockets** auf das UDP und **Raw Sockets** auf das IP Protokoll auf (im Fall der TCP/IP Protokollfamilie). Der Typ der Sockets wird bei der Systemgenerierung festgelegt.
Eine Socketadresse für das TCP/IP Protokoll besteht aus der IP Adresse und der Portnummer. Die IP Adresse ist eine 32-Bit Zahl, die normalerweise durch vier Dezimalzahlen, getrennt durch einen Punkt, dargestellt wird. Der Port ist der Eingangspunkt zu einer Applikation und wird durch eine 16-Bit Integer Zahl dargestellt. Ports sind normalerweise Eingangspunkte zu Service der Server Applikationen. Wichtige kommerzielle Server Programme (z.B. Oracle DBMS, FTP) haben ihre festgelegten, sogenannten *well-known* Ports.

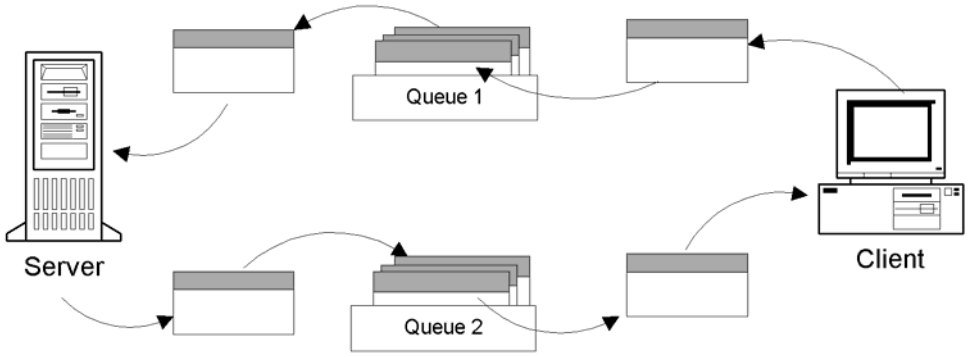
2. Remote Procedure Call (RPC)
Beim RPC ruft ein Client Prozess eine Funktion auf einem entfernten Server auf und verbleibt im Wartezustand bis er eine Antwort erhält - der RPC arbeitet also *synchron*. Parameter werden wie bei jeden gewöhnlichen Prozeduraufruf übergeben.
Es ist ein Client- und ein Serverteil des RPC nötig. Der Clientteil (Client-Stub) wird von der Client Applikation wie eine lokale Prozedur aufgerufen. Der auszuführende Prozedurcode befindet sich jedoch auf dem Server. Also werden die Parameter in eine Nachricht verpackt und an den Serverteil des RPC (den Server-Stub) gesendet. Dieser packt dann die Parameter wieder aus, führt mit diesen Parametern den Prozedurcode aus und sendet die Ergebnisse, wiederum in einer Nachricht verpackt, zurück zum Client-Stub. Die Rückgabedaten werden hier wieder ausgepackt und an die aufrufende Applikation übergeben.



Die Schnittstelle des RPC muss in der Interface Description Language (IDL) deklariert werden, damit das Ein- und Auspacken korrekt funktioniert. Ein IDL-Compiler erzeugt daraus dann die Client- bzw. Server-Stubs. Die Implementierung der Prozedurrümpfe erfolgt wieder wie bei normalen Prozeduren.
Obwohl RPC das Leben eines Programmierers bereits erheblich vereinfachen gibt es einige Schwierigkeiten und Fehlerquellen zu beachten:
Wenn sehr viele Clients auf die Serverfunktionen zugreifen wird schnell ein großer Anteil der Rechenleistung des Servers nur für das Starten und Stoppen des Service, für die Priorisierung der Anfragen, Sicherheitsabfragen und Load-Balancing verbraucht.
Wie wird bei Fehlern reagiert? Ein recht umfangreicher Punkt, da beide Seiten separat ausfallen können. Es ist daher wichtig, dass die eingesetzte Software alle auftretenden Fehlerkombinationen abfangen kann. Wenn der Server nicht antwortet, blockiert der Client normalerweise. Nach einem Timeout muss er seine Anfrage wiederholen. Stürzt der Client nach einer Anfrage ab, muss der Server alle bis dahin vorgenommenen Veränderungen rückgängig machen können u.s.w.
Der Server muss garantieren, dass die gleichen Anfragen nur einmal abgearbeitet werden. Dies ist insbesondere bei mehreren Servern wichtig, wenn die Anfrage vielleicht an den zweiten Server abgesetzt wird, wenn der erste nicht reagiert.
Um mit RPC eine sichere und atomare Ausführung von Funktionen auch unter der Berücksichtigung aller möglichen Fehler zu gewährleisten ist ein erheblicher Entwurfs- und Implementierungsaufwand erforderlich.

3. Message Oriented Middleware (MOM)
Mit Hilfe von MOM Nachrichten und Warteschlangen können Clients und Server über ein Netzwerk kommunizieren, ohne dass sie über eine spezielle Verbindung fest miteinander verbunden sind- also *asynchron*. Clients und Server können so zu unterschiedlichen Zeiten laufen. Man

kommuniziert einfach, indem man Nachrichten in eine Warteschlange stellt bzw. Nachrichten aus der Warteschlange entnimmt.



Dieses Prinzip löst alle Probleme, die beim RPC oder bei der peer-to-peer Verbindung aufgetreten sind:

- Der Client kann immer noch eine Anfrage senden, auch wenn der Server beschäftigt, runtergefahren oder gar nicht angeschlossen ist. Es muss nur die Warteschlange verfügbar sein, in die der Client seine Anfrage ablegen kann. Ist der Server wieder verfügbar, kann dieser dann die Nachrichten aus der Schlange abholen und bearbeiten.
- Umgekehrt kann auch der Server eine Antwort senden, wenn der Client runtergefahren oder nicht angeschlossen ist. Es muss wiederum nur seine Antwort- Warteschlange verfügbar sein. Der Server setzt seine Antwort einfach in diese Schlange. Ist der Client wieder verfügbar, prüft er in seine Warteschlange und findet die erwartete Antwortnachricht des Servers.
- Mehrere Server können ihre Anfragen der gleichen Warteschlange entnehmen und so die Arbeitslast auf diese verteilen. Sobald ein Server eine Anfrage abgearbeitet hat kann er die nächste aus der Schlange entnehmen und diese abarbeiten. Es kann also nicht passieren, dass ein Server überlastet ist, während sich der andere im Leerlauf befindet.
- Anfragen in der Warteschlange können für eine prioritätsgesteuerte Bearbeitung mit Prioritäten versehen werden.

1.5.3.2 Client-Server-Architekturen

1.5.3.2.1 Architekturformen

In der Praxis haben zahlreiche Realisierungen von Client-Server-Architekturen ergeben. Sie entsprechen weitgehend den bekannten Verteilungsformen[91] von Anwendungssystemen.

1. Disk-Server

Anforderungen des Dateisystems werden an den Disk-Server weitergeleitet und dort auf die Plattenperepherie des Servers umgesetzt

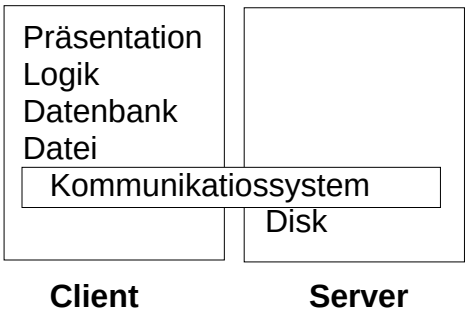


Abb. 1.5-5: Disk-Server

Es gibt nur einen physikalischen Plattenspeicher, der von allen Clients benutzt werden kann. Eine spezielle Anwendung des Disk-Servers ist der "Remote-Boot" für Arbeitsplatzrechner ohne Diskettenlaufwerk (diskless client). Das Betriebssystem wird dabei vom Disk-Server geladen.
Bei einem Disk-Server können zwar zahlreiche Leser, aber nur ein einziger Schreiber auf die gemeinsam genutzte Ressource zugreifen. Das ist für das Laden von Programmen oder das Lesen von gemeinsamen Daten ausreichend.

2. File-Server

Der File-Server empfängt alle Dateianforderungen, die von einzelnen Clients (z.B. PCs und Workstations) an ihn gesandt wurden.

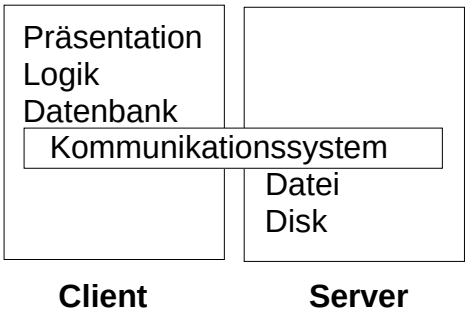


Abb. 1.5-6: File-Server

[91] Hans Joachim Petzold, Hans Jochen Schmitt: Verteilte Anwendungen auf der Basis von Client-Server-Architekturen in HMD 170/1973, S. 79 - 91

Operationen, z.B. das Sperren von Bereichen werden zentral im File-Server verwaltet. Mehrere Anwendungen können konkurrierend auf gemeinsame Datenbestände zugreifen. Es gibt u.U zahlreiche unabhängige Datenbanksysteme jedoch nur ein Dateisystem. Die einzelnen Workstations sind mit dezentraler Intelligenz ausgestattet, so daß alle Programme im lokalen Arbeitsspeicher ablaufen. Zur Inanspruchnahme bestimmter Dienste werden Programmaufrufe über die in einer Workstation eingebaute Netzwerkkarte an den Server weitergeleitet.
Nach dem Prinzip des File-Server arbeiten die meisten Netzwerkbetriebssysteme. Bekanntestes Produkt ist zur Zeit Novell Netware. Mit Hilfe sog. Netware Loadable Moduls (NLM) läßt sich das Netzwerk an verschiedene Architekturen[92] anpassen.
Den restlichen Markt für Netzwerkbetriebssysteme teilen sich LAN-Manager bzw. LAN-Server[93] und Banyan Vines.

Netware File Server

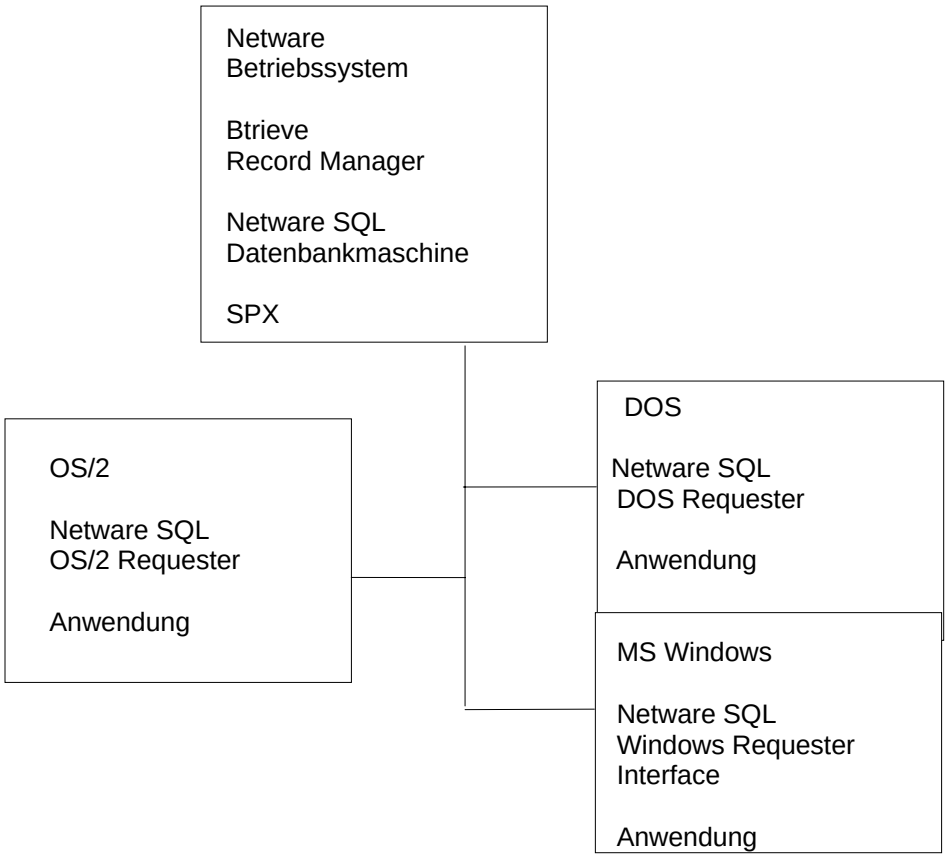


Abb. 1.5-7: Client-Server Architektur des Netware SQL

Btrieve ist die Zugriffsmethode von Netware SQL. Geeignete „client requester"-Programme kommunizieren über SPX (Sequenced Paket Exchange Protocoll) mit Netware SQL.
Auch UNIX-Server sind (auch in PC-Netzen) weit verbreitet. Für alle UNIX-Abkömmlinge gibt es NFS (Network File System)[94]. Es basiert auf dem TCP/IP-

[92] Viele Systemdienste wie etwa die ganze Schichtung möglicher Netzwerkprotokolle (IPX, ZCP/IP usw.) wurden auf der Server-Seite als NLM realisiert
[93] Der LAN-Manager kommt von Microsoft, der LAN-Server von IBM. Der Ansatz ist aber gleich. Der Server-Teil setzt auf OS/2 als Betriebssystem. Nachfolger des LAN-Manager ist inzwischen der Windows NT Advanced Server, der Netzwerkfunktionen von Windows NT benutzt

Netzwerkprotokoll. Heterogene Netzwerke bieten in diesem Zusammenhang keine grundsätzlichen Probleme. Wegen der in UNIX-Betriebssystemen enthaltenen unbegrenzten TCP/IP- und NFS-Lizens kann das Netzwerk beliebig wachsen.

3. Datenbank-Server

Datenanforderungen werden bereits an der Schnittstelle zum DB-System (z.B. in Form von SQL-Anweisungen) abgefangen. Das Datenbanksystem mit allen untergeordneten Schichten befindet sich auf dem Server.

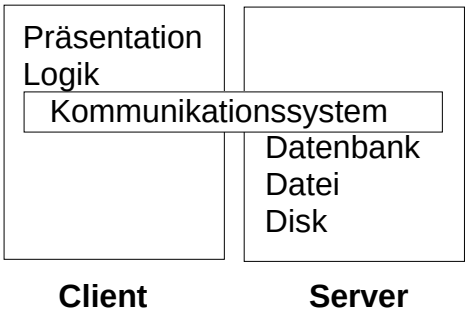


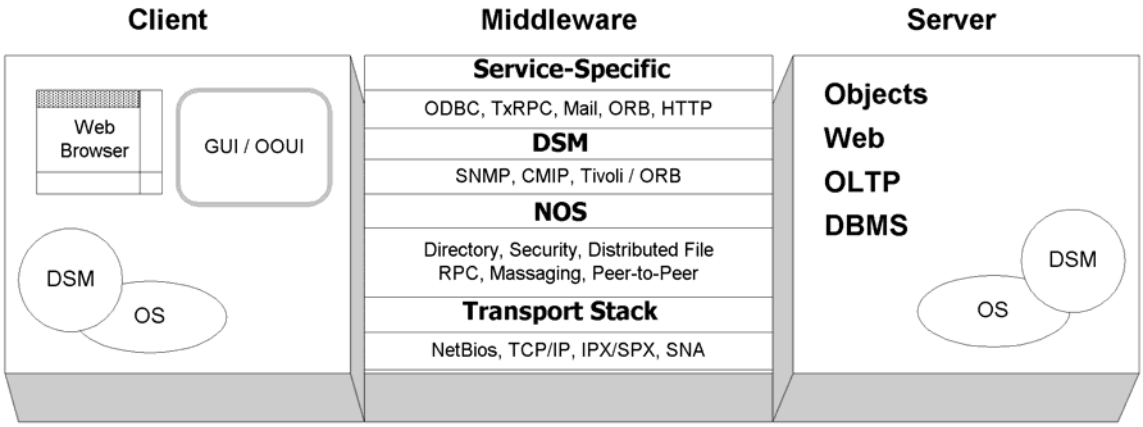
Abb.1.5-8 Datenbank-Server

Bei der Datenverarbeitung im Netzwerk werden Programme (, die lokal im Netzwerk zur Verfügung stehen,) in den Hauptspeicher des Client geladen und ausgeführt. Erhalten diese Programme Datenbankzugriffe, dann sorgt eine spezielle Softwareschicht, die „**Middleware**“[95], dafür, daß alle Zugriffe auf einen Datenbankserver „umgelenkt“ werden. Im Rahmen der 3-Tier-Architektur wird das Client/Server-Modell in drei große Blöcke unterteilt: Client, Middleware und Server.
Im Client Block läuft der Client Teil der Applikation auf einem Betriebssystem, welches ein Graphical User Interface (GUI) oder ein Object Orientet User Interface (OOUI) bietet und Zugriff zum verteilten Service hat. Das Betriebssystem arbeitet mit der Middleware zusammen und überlässt ihr die Bearbeitung der nicht lokalen Servicees. Außerdem läuft auf dem Client eine Komponente des Distributed System Management (DSM).
Im Server Block läuft die Server Seite der Applikation. Die Serversoftware läuft meist auf einem kompletten Server Software Package. Die wichtigsten Serverplattformen sind dabei: SQL Datenbank Server, TP Monitore, Groupware Server, Objekt Server und Web Server. Das Betriebssystem stellt das Interface zur Middleware bereit und liefert die Anfragen für Service. Auch auf dem Server läuft eine Komponente des DSM.
Der Middleware Block läuft auf der Client- und der Serverseite der Applikation und kann wieder in drei große Kategorien unterteilt werden: Transport Stacks, Netzwerk Operating System (NOS) und Service-spezifische Middleware. Auch der Middleware Block hat eine DSM Komponente.

---

[94] eine Entwicklung von Sun Microsystems, es gibt auch ein PC-NFS von Sun

[95] Middleware ist keine eigenständige Software. Sie muß auf unterschiedliche Art und Weise in neue Anwendungen integriert werden, z.B. als Webserver Plug-In bzw. Add On, als Java-Klassen, Perl-Module, Java-Beans ider Servlets. Middleware-Produkte werden zum Transfer von Daten zwischen Datenbanken ind anderen (Client-) Anwendungen.



Middleware ist ein recht ungenauer Begriff, der all die verteilte Software beinhaltet, die benötigt wird um Interaktionen zwischen Client und Server zu ermöglichen; oder anders gesagt: Ist die Verbindung, die dem Client ermöglicht einen Service vom Server zu erhalten.
Middleware beginnt beim Client API, über welchen ein Serviceaufruf abgesetzt wird, beinhaltet die Übertragung des Aufrufes über ein Netzwerk und die resultierende Antwort auf den Aufruf. Sie beinhaltet nicht die Software die den eigentlichen Service bietet (das befindet sich im Server Block) oder das User Interface (welches sich im Client Block befindet).
Der Datenbankserver berechnet das Ergebnis und das Programm auf dem Client wird automatisch mit diesen Daten versorgt. Im Unterschied zum Dateiserver führt der Datenbankserver komplexe Operationen aus.
Eine Transaktionsverwaltung ist über den Datenbank-Server möglich. Es können auch Server eingesetzt werden, die eine andere Betriebssystem- oder Hardwarearchitektur haben als Client-Rechner. So sind spezielle Datenbankrechner als Datenbank-Server einsetzbar, sobald sie über ein Kommunikationssystem von den Clients aus erreicht werden können. Bekannte Produkte sind: Oracle, Ingres, IBM Database Manager sowie Server von SyBase und Gupta Technologies.

Das Oracle Client-/Server-Konzept
Ab Version 7 heißt das relationale Datenbanksystem der Firma Oracle „Oracle7Server“. Dieser Begriff betont ab Version 7 ausgeprägte Serverfähigkeiten der Datenbankssoftware von Oracle.
Auch dieser Datenbankserver muß in irgendeiner Form (mit SQL-Befehlen) versorgt werden. Dies übernimmt ein Client-Prozeß, der auf dem gleichen Rechner läuft wie der Oracle7Server (lokale Anwendung) oder auf einem anderen Rechner im Netzwerk (Client-/Server-Anwendung).
Eine wichtige Aufgabe des Client ist die Identifikation der Anwendung, die ausschließlich über das User Programmatic Interface (UPI) an den Datenbankkern auf den Server gesandt wird.

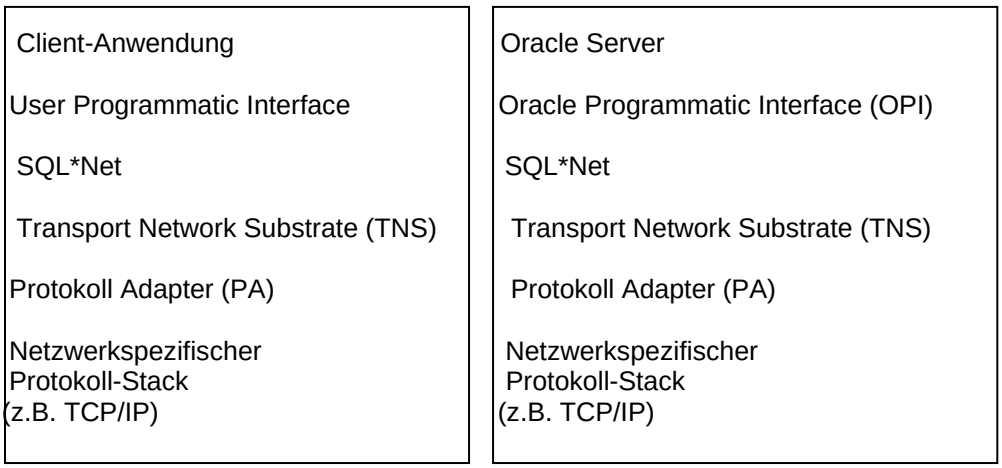


Abb. 1.5-9: Die Komponenten bei einer Client-/Server-Verbindung

UPI ist eine Schicht der Client-Anwendung mit einem Satz von Funktionen, über die ein SQL-Dialog zwischen Client und Server durchgeführt werden kann. Die Kontrolle wird dann an SQL*Net weitergegeben, das für den Transport der Daten zuständig ist. In Client-/Server-Verbindungen sorgt **SQL*Net** für die Verbindung zwischen Client und Server.
SQL*Net ist die Middleware von Oracle. Auf der Client-Seite besteht SQL*Net im wesentlichen aus einigen Bibliotheken und Konfigurationsdateien. Auf der Server-seite gibt es zusätzlich einen Listener-Prozeß, der auf Verbindungen wartet und entsprechende Verbindingen aufbaut. Das Interface der Anwendungen und des Oracle7Server zu SQL*Nert ist protokollunabhängig.
SQL*Net ist schichtenweise strukturiert. Die unterste Schicht bilden netzwerkspezifische Protokolle, z. B. TCP/IP, IPX/SPX. Darauf folgt die „Protocol Adapter (PA) – Schicht“. Auf diese Schicht baut die zentrale Komponente, Transport Network Substrate (TNS) auf. TNS realisiert an seiner Schnittstelle elementare Kommunikationsfunktionen und gibt seine Informationen an die Protokoll Adapter weiter.
TNS soll eine einheitliche Schnittstelle erzeugen. Applikationen, die auf TNS aufsetzen, können völlig unabhängig von spezifischen Protokollen implementiert werden. Auf TNS setzen zur Zeit zwei Produkte auf: SQL*NET und das Multiprotocoll Interchange. SQL*NET ist die Schnittstelle auf die Clients bzw. Server aufsetzen. Das Multiprotocol Interchange dient zur Protokollumwandlung und ermöglicht, daß Client und Server mit unterschiedlichen Protokollen betrieben werden können.

Im Bereich des Kommunikationsteils auf dem Client gibt die Software-Komponente SQL*Net[96] alle notwendigen Informationen an TNS, das Daten an den Prokoll-Adapter (PA) weiterleitet, der für den protokollspezifischen Transport zuständig ist. Das Netzwerk-Protokoll transferiert die Daten über das physikalische Netz zum Server. Dort angekommen nehmen die Daten den umgekehrten Weg über PA, TNS und SQL*Net. Das Multiprotocoll Interchange des Server verfügt über eine Listener (TNS-), der ankommende Verbindungen auf das Ziel überprüft und ann einen Serverprozeß zur Abarbeitung der SQL-Anweisungen startet. Von der SQL*Net-Schicht werden Daten an das Oracle Programmatic Interface (OPI) weitergegeben (arbeitet entgegengesetzt zum UPI). Für jeden Aufruf des UPI gibt es im OPI eine Funktion, die die angeforderte Aufgabe erfüllt. Oberhalb des OPI setzt der Server auf, der die Anforderungen des Client erfüllt und die Ergebnisse über das OPI an den Client zurücksendet.

SQL-Server

SQL-Server bauen auf der "Client-Server-Architektur" auf. Der Datenbankserver (database server) läuft auf einem eigens dafür vorgesehenen Rechner und bietet die notwendigen Datenbankfunktionen für alle Netzteilnehmer an. Der Datenbankserver ist auf die Sprache **SQL** ausgerichtet. Der Client fordert mit einer SQL-Anfrage eine Ergebnistabelle an, die er dann weiterverarbeitet. Der Server ist in der Lage, SQL-Anfragen auszuwerten und ganze Tabellen zu übergeben.
SQL-Server haben einen "transaction manager" zur Synchronisation von Tabellen und Indexverzeichnissen (z.B. bei einem Absturz). Dauerhaft enthalten Tabelle und Indexverzeichnisse die Resultate von ausschließlich abgeschlossenen Transaktionen. "Backup/Restore-Utilities" generieren Kopien und sorgen für die Wiederherstellung der Datenbank (z.B. bei Plattenausfall). Die vollständige Wiederherstellung aller Aktualisierungen zwischen der letzten Sicherung (Backup) und dem Zeitpunkt des Plattenfehlers übernehmen "Recovery-Prozeduren". SQL-Server unterstützen Entitäts- und Beziehungsintegrität [97] , automatische Seiten- oder Datensatzsperre und verfügen uber eine automatische Erkennung von Verklemmungen.
Der Server erledigt die Arbeit, der Front-End zeigt Benutzerfreundlichkeit. Front-Ends von Tabellenkalkulationen oder "Multimedia-Programme" sollen auf verschiedene DBMS zurückgreifen können. Ein genormtes SQL stellt (zumindestens theoretisch) sicher, daß die Kommunikation zwischen beiden reibungslos funktioniert. In der Praxis verlangt jedoch jedes DBS eine Sonderbehandlung [98].

[96] Der Begriff SQL*Net wird für zwei Dinge verwendet: Einerseits als Oberbegriff für alle Netzwerkprotokolle und die Middleware von Oracle, andererseits als Bezeichnung für diejenige Softwarekomponente, die auf Client- und Serverseite installiert ist.
[97] vgl. 1.2.6
[98] vgl. ODBC (Open Database Connectivity)-Interface von Microsoft, das im wesentlichen eine Windows-API ist. Für jedes DBMS gibt es einen ODBC-Treiber.

4. Präsentations-Server

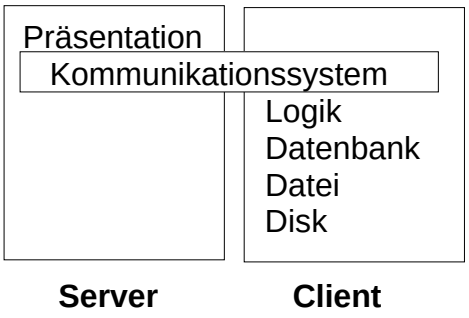
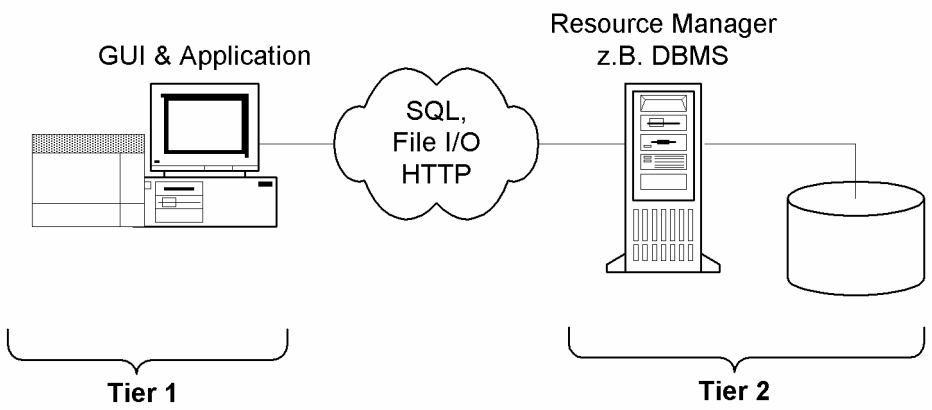


Abb. 1.5-10: Präsentationsserver

Diese Form der Anwendung wird auf Unix-Systemen mit dem X-Window-Ansatz[99] verfolgt. Der den Dienst der Präsentation anbietende Arbeitsplatzrechner ist hier der Server. Der Client bedient sich dieser Dienste zur Interaktion mit dem Benutzer.

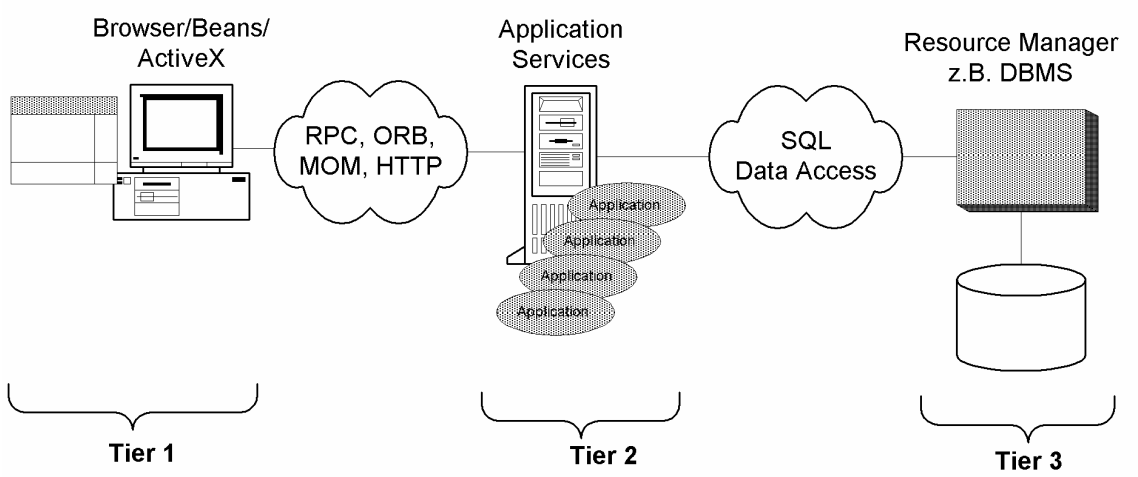
1.5.3.2.2 2-Tier und 3-Tier Client-/Server-Architekturen

Bei **2-Tier Client/Server Systemen** befindet sich die Applikationslogik entweder im User Interface des Clients oder in der Datenbank des Servers.
Beispiele für 2-Tier Client/Server Architekturen sind: File Server und Datenbank Server mit gespeicherten Prozessen.



Bei **3-Tier Systemen** befindet sich die Applikationslogik in der Mittelschicht, getrennt von den Daten oder dem User Interface. Sie kann somit getrennt vom GUI und der Datenbank gewartet und entwickelt werden.
Beispiele für 3-Tier Client/Server Architekturen sind: TP-Monitore, Object Transaction Monitore, Web Application Server.

[99] X-Window-Ansätze realisieren die Window-Manager OSF-Motif, Open Look



Vergleich von 2- und 3-Tier-Architektur: Der größte Vorteil der 2- Tier Architektur ist, dass Client/Server Applikationen sehr einfach und schnell erstellt werden können, z.B. mittels visuellen Programmiertools. Diese Applikationen arbeiten meist noch recht gut in Prototypen und kleineren Installationen, zeigen aber schnell ihre Grenzen wenn sie in größeren Anwendungssystemen mit einer hohen Benutzerzahl und vielen konkurrierenden Zugriffen eingesetzt werden.
3- Tier Architekturen sind besser skalierbar, robust und flexibel. Sie können Daten von mehreren verschiedenen Quellen verarbeiten, sind im Netzwerk einfacher zu administrieren und zu entwickeln da der meiste Code auf dem Server läuft. 3- Tier Applikationen verringern die Netzwerklast durch zusammengefasste Serviceaufrufe. Anstatt mit der Datenbank direkt zu kommunizieren wird mit der Mittelschichtlogik auf dem Server kommuniziert. Diese arbeitet dann eine Reihe von Datenbankzugriffen ab und reicht nur das gewünschte Ergebnis zurück. Daraus resultiert eine höhere Performance durch wenige Serveraufrufe.
Außerdem wird eine höhere Sicherheit gewährleistet, da kein direkter Zugriff auf die Datenbank durch den Klienten erfolgt.

1.5.3.3 Client/Server und Internet / Intranet

Grundlage der Client-/Server-Architektur ist eine Verteilung zwischen verschiedenen Systemen und Systemteilen. Die Internet-Technologie[100] ist eine Hardware- und Systemstruktur, kombiniert mit einheitlichen Protokollen und Präsentationsformaten. Sie repräsentiert eine geschickte Art der Praktizierung von Client-/Server-Architekturen.

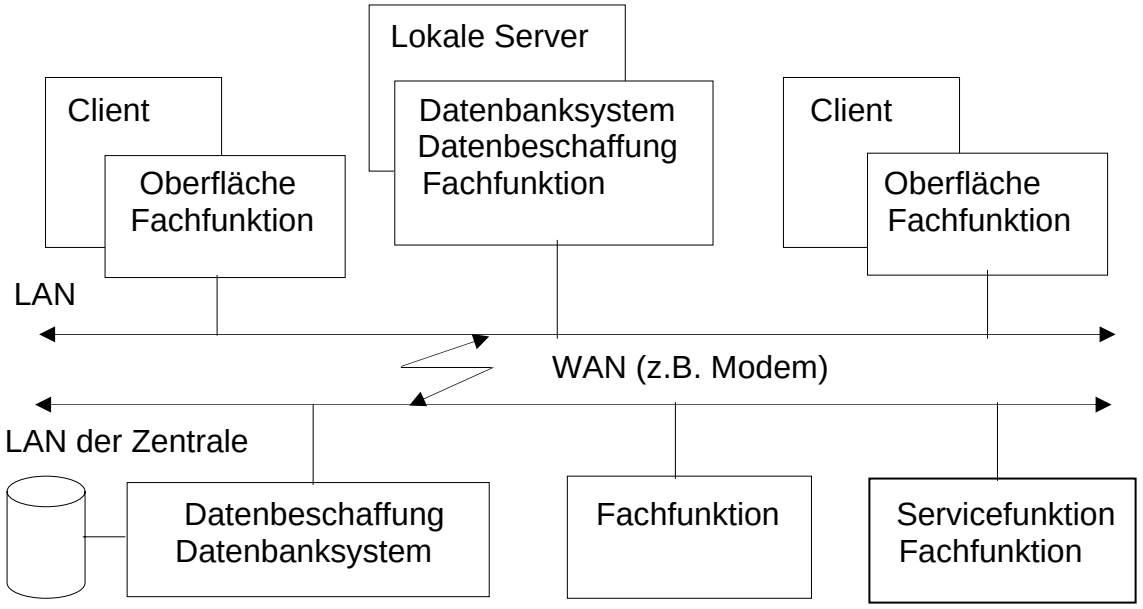


Abb. 1.5-11: Aufgabenverteilung auf verschiedenen Schichten (Präsentation, Funktionen, Datenbeschaffung / Datenbank, Services) des Client-/Server-Modells

Die Abb. zeigt die Verteilung der Aufgaben auf verschiedene Schichten:
Die Präsentationsschicht übernimmt die Darstellung der Information gegenüber dem Benutzer (klassische Frontend-Bearbeitung). Die Funktionsschicht umfaßt die fachlichen Funktionen, unabhängig von Darstellung und Speicherung. Die Datenbeschaffungsschicht umfaßt die Speicherung der Daten auf den erforderlichen Medien mit den entsprechenden Systemen. Die Service-Schicht stellt Werte und Funktionen in allgemeiner Form bereit, die systembezogen (oder umgebungsbezogen)[101] sind.
Zur Nutzung des Internet steht eine Vielzahl von Anwendungen zur Verfügung, die als Internet-**Dienste** bezeichnet werden. Die Dienste[102] stützen sich auf ein Client-/Server-Modell ab. Bekannte Dienste im Internet sind: Telnet, File Transfer Protocol (FTP), Usenet News, World Wide Web (WWW).
Alle Internet-Dienste stützen sich auf das Client-/Server-Konzept. Das Client-Programm (Client) stellt die Schnittstelle zwischen Benutzer und Server-Programm (Server) dar. Der Server bietet Informations- und Kommunikationsvermittlung an. Aufgabe des Client ist es, Anfragen des Benutzers in eine maschinenverständliche Art „umzuformulieren“ und dem Benutzer die vom Server gelieferte Antwort zu präsentieren. Für die Benutzung eines Internet-

[100] firmenintern als Intranet genutzt
[101] Das fängt beim Tagesdatum an und reicht zu so komplexen Funktioen wie bspw. Zeitsynchronisierung innerhalb eines komplexen Netzes oder der Nachrichtenvermittlung zwischen Objekten (Object Request Broker.
[102] Für eine Vielzahl von Diensten sind Standqardisierungen in sog. RFCs (Request for Comments) niedergelegt

Dienstes ist ein Client- und ein Server-Programm nötig. Der Client übernimmt Vorfeldaufgaben (front-end application) der Informationsverarbeitung und erlaubt unterschiedliche Datenquellen unter einer Oberfläche zu integrieren.
Internet-Dienste ermöglichen die Kommunikation mit anderen Internet-Teilnehmern, die Nutzung von Informationsressourcen im Internet und das Anbieten von Informationen über das Internet. Der Internet-Dienst, dem das Internet das exponentielle Wachstum verdankt, ist das World Wide Web (WWW).
Die Idee des **World Wide Web** (WWW oder einfach nur Web) ist Anfang 1989 am CERN[103] entstanden. Auslöser waren Schwierigkeiten beim Auffinden relevanter Informationen im Internet in Hard- und Software. Zur Lösung des Problems wurde ein auf der Client-/Server-Architektur aufbauendes, hypertext-basiertes System vorgeschlagen. Hypertext wird dabei als Weg verstanden, Informationen derart miteinander zu verknüpfen, daß sich der Benutzer nach eigenem Belieben durch die Informationen bewegen kann. Die Verknüpfungen erscheinen dabei als Knoten in einem verwobenen Netz.

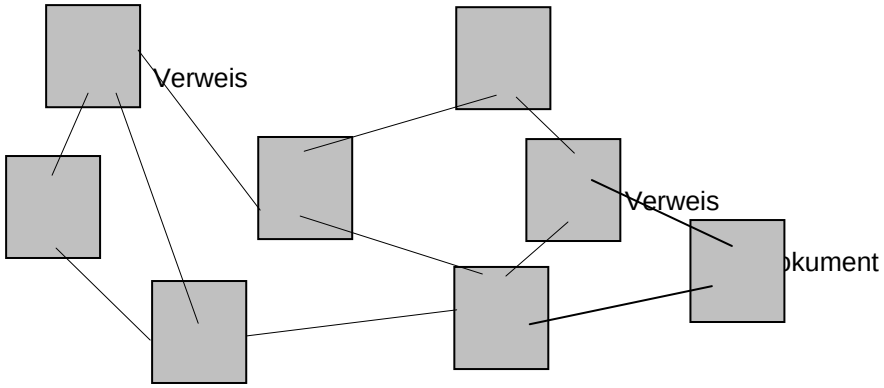


Abb. 1.5-12: Struktur des World Wide Web

Eine verteilte Hypermedia-Anwendung kann als Netz angesehen werden, dessen Knoten Text, Grafik, Ton darstellen. Verbindungen (Links) kennzeichnen die Beziehungen zwischen den Knoten und werden von der Systemsoftware verwaltet. Die möglichen Verbindungen werden im Dokument (Fenster) an der entsprechenden Stelle gekennzeichnet. Der Benutzer kann durch Anklicken derartiger Markierungen im Dokument oder durch Weiterblättern bzw. Rücksprung zu einem früheren Knoten fortfahren, solange er will.
Dokumente werden in einem speziellen Format, dem Hypertext-Format gespeichert. Zum Erstellen von Dokumenten steht die Seitenbeschreibungssprache HTML (Hypertext Markup Language) zur Verfügung[104].
Die wichtigsten HTML-Steueranweisungen sind Hyperlinks. Damit kann man auf Dokumente verweisen, die irgendein anderer Rechner im Internet bereitstellt. Diese Verweise bilden ein weltumspannendes Netz, das WWW.
Das **Hypertext Transfer Protocol** (HTTP) dient zur Übertragung von Informationen aus dem World Wide Web (WWW). HTTP ist ein objektorientiertes Protokoll zur einfachen Übertragung von Hypertext-Dokumenten zwischen Client und Server.
Client-Programme, die HTTP benutzen, werden (in der Regel) als **Web-Browser**, Server-Programme **als Web-Server** bezeichnet. Der Browser schickt an den

[103] europäisches Zentrum für Teilchenphysik bei Genf
[104] bezieht sich auf die in ISO 1986 standardisierte SGML (Standard Generalized Markup Language).

Server die Aufforderung eine bestimmte HTML-Seite zu übertragen. Falls er in dieser Seite weitere Verweise (z.B. auf Bilder) entdeckt, schickt er weitere Übertragungswünsche hinterher. Das Besorgen der gewünschten Dokumente erfolgt über ein einheitliches Adreßschema, dem Uniform Resource Locator (URL), durch den Internet-Standort und die Art der zu übertragenden Information identifiziert werden. Der URL besteht im wesentlichen aus zwei Teilen:

- Er wird mit der Bezeichnung des Übettragungsprotokolls eingeleitet, z.B.: „http“.
- Danach folgt (mit zwei vorangestellten //) der Name des Internet-Rechners, auf dem die gewünschte Information gespeichert ist. Wird eine bestimmte Datei in einem bestimmten Verzeichnis referenziert, dann werden noch Verzeichnisname und Dateiname angefügt.

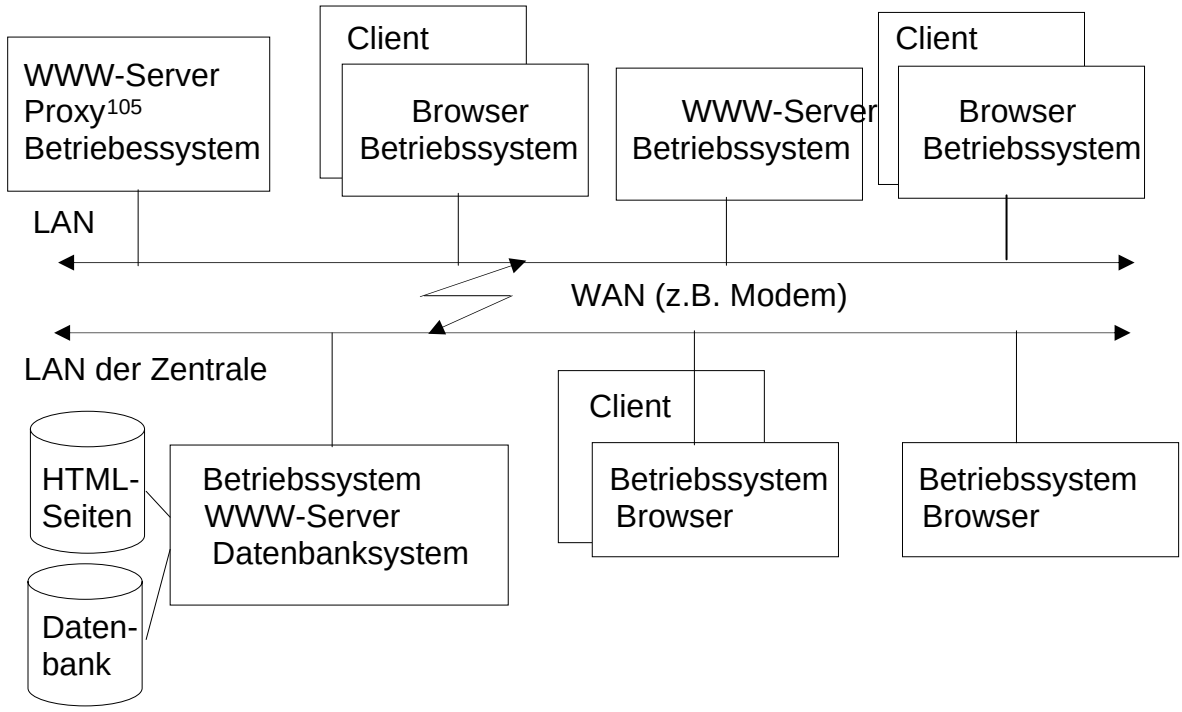


Abb. 1.5-13: Intranet-Architektur

Bei der Client-/Server-Kommunikation über das Internet benötigt man einen Server, der ständig in Bereitschaft ist und auf Anfragen von einem Client wartet. Der Client baut eine Verbindung auf, indem er eine Anfrage an den Server richtet und dieser die Verbindung aufnimmt. Das einfachste Bsp. hierfür ist ein Web-Server, der auf einem Host-Rechner installiert ist. Der Web-Browser fungiert dabei als Client, der eine Verbindung zu einem bestimmten Web-Server für den Datenaustausch herstellt. In erster Linie wird der Client HTML-Dokumente vom Web-Server empfangen.
Viele Web-Knoten legen jede Web-Seite als eigene HTML-Datei ab. Solche Seiten sind allerdings statisch, d.h. sie erzeugen bei jedem Aufruf denselben Inhalt. In vielen Unternehmen steigt aber das Interesse, vorhandene, in Datenbanksytemen gehaltene Dokumente und Informationen für Web-Clients verfügbar zu machen. Die Anbindung einer Datenbank an einen Web-Server erleichtert die

[105] Proxy-Server: Das ist ein zusätzliches lokales Datenbeschaffungsinstrument für einen bestimmten anwenderkreis (Clients des oberen LAN). Die Clients dieses Anwenderkreises gehen nicht direkt ans Netz, sondern beauftragen ihren Proxy über die schnelle lokale Verbindung, eine oder mehrere Informationsseiten zu beschaffen. Der Proxy holt sich die Seite, speichert sie in einem lokalen Cache-Speicher und liefert sie an den lokal anfordernden Client aus.

Aktualisierung der Inhalte. Allerdings müssen dazu die Daten mediengerecht aufbereitet, d.h. in HTML, der Seitenbeschreibungssprache des Web verpackt werden. Das geschieht mit Hilfe von Programmen, die Daten aus der Datenbank auslesen und aus den so gewonnenen Informationen HTML-Dokumente erzeugen. Die zugehörigen Programme laufen auf dem Web-Server, von dem auch die statischen Dokumente geladen werden.

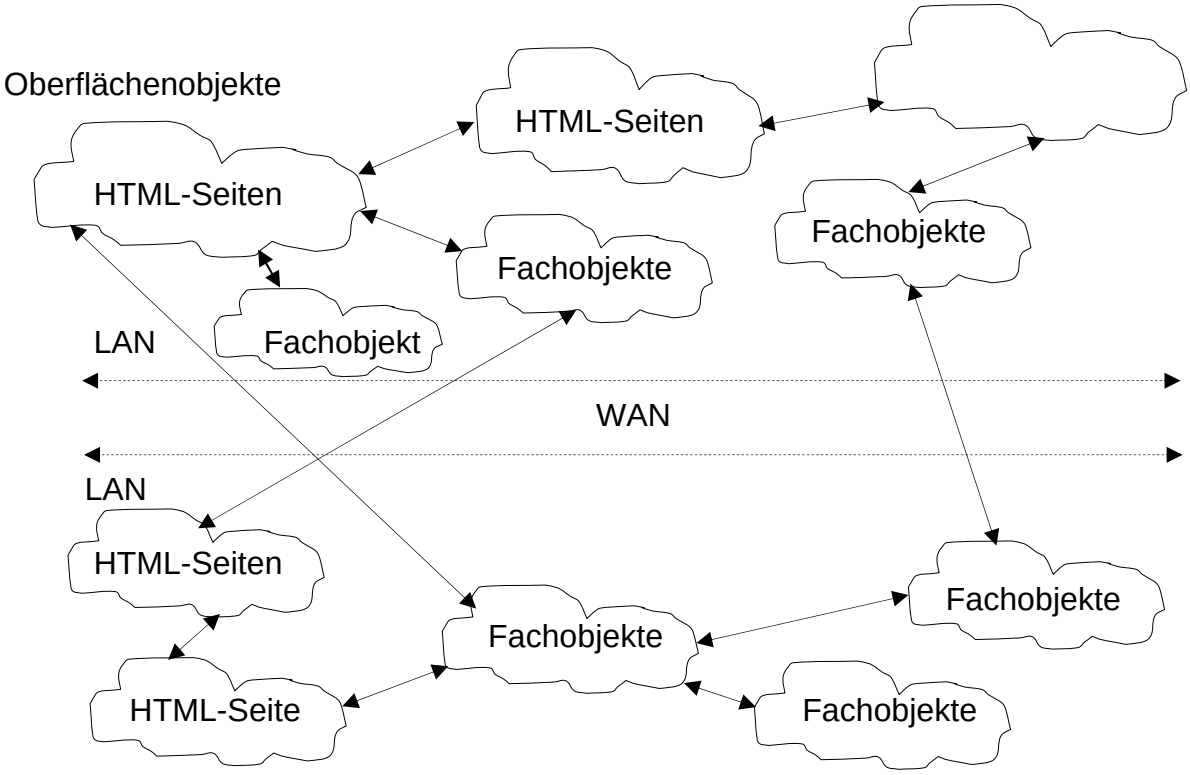


Abb. 1.5-14: Aufbau und Kommunikation der Objekte im Internet

Typischerweise werden Applikationen in 3 Aufgabenbereiche unterteilt: Datenhaltung, Anwendung(slogik) und Präsentation[106]. Diese 3 Aufgabenbereiche müssen nicht auf einem Rechner oder von einem Prozeß wahrgenommen werden, sondern können auf mehrere Rechner bzw. Prozesse verteilt sein.

[106] Vgl. Client-Server-Architekturen

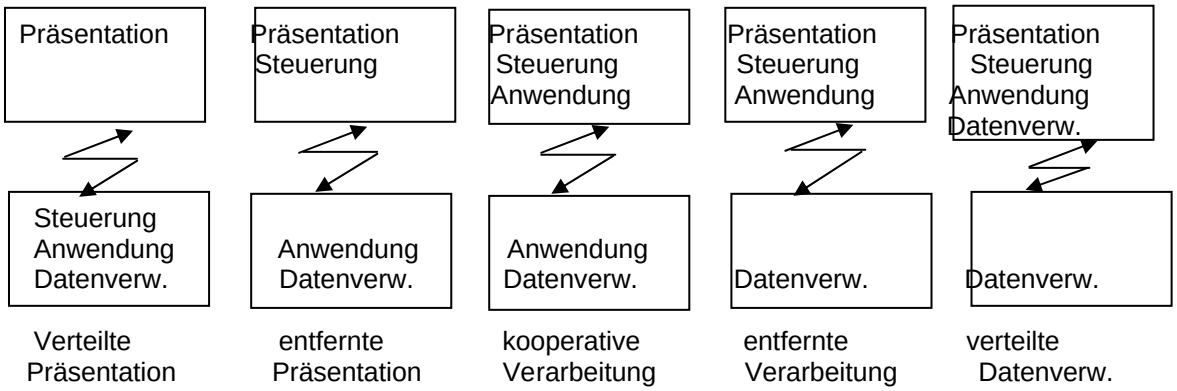
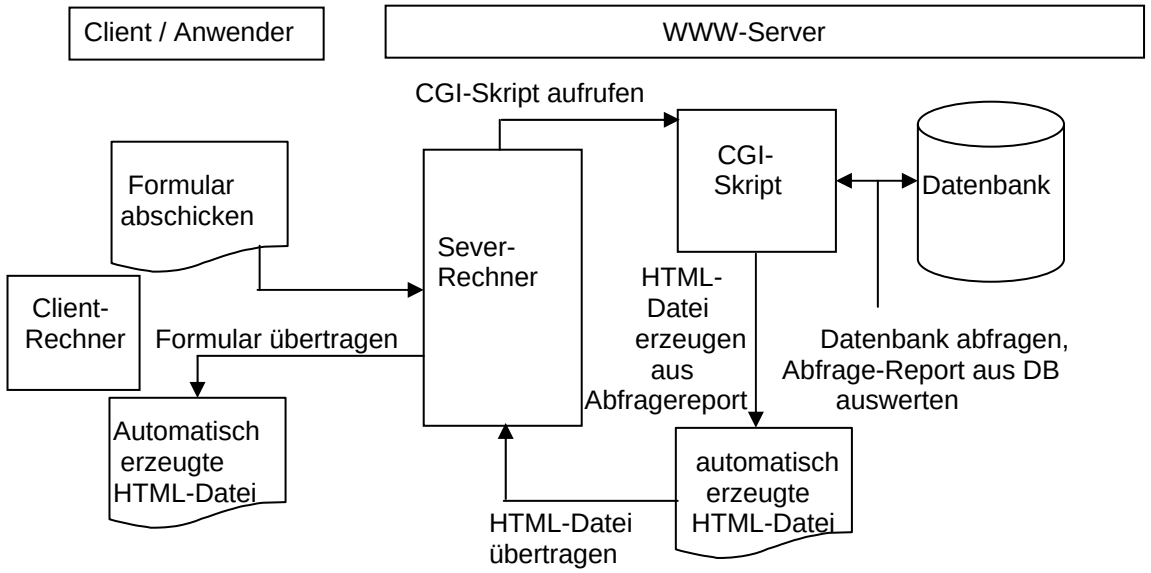


Abb. 1.5-15: Verteilungsmöglichkeiten im Client-/Server-Modell

Bei der **entfernten Repräsentation** (z.B. Common Gateway Interface (`CGI`)) findet die Verarbeitung „Server"-seitig statt, auf dem Client werden nur Repräsentationsaufgaben ausgeführt:
Das Common Gateway Interface (Allgemeine Vermittlungsrechner-Schnittstelle) ist eine Möglichkeit, Programme im WWW bereitzustellen, die von HTML-Dateien aus aufgerufen werden können, und die selbst HTML-Code erzeugen und an einen WWW-Browser senden können. CGI umfaßt Programme, die auf einem Server-Rechner im Internet liegen und bei Aufruf bestimmte Daten verarbeiten. Die Datenverarbeitung geschieht auf dem Server-Rechner. CGI-Programme können auf dem Server-Rechner Daten speichern, zum Beispiel, wie oft auf eine WWW-Seite zugegriffen wurde. Bei entsprechendem Aufruf kann ein CGI-Programm gespeicherte Daten auslesen und daraus HTML-Code generieren. Dieser "dynamisch" erzeugte HTML-Code wird an den aufrufenden WWW-Browser eines Anwenders übertragen und kann dort individuelle Daten in HTML-Form anzeigen, zum Beispiel den aktuellen Zugriffszählerstand einer WWW-Seite.
Die sogenannte CGI-Schnittstelle muß von der WWW-Server-Software unterstützt werden[107]. Es gibt keine Vorschriften dafür, in welcher Programmiersprache ein CGI-Programm geschrieben ist. Damit das Programm auf dem Server-Rechner ausführbar ist, muß es entweder für die Betriebssystem-Umgebung des Servers als ausführbares Programm kompiliert worden sein, oder es muß auf dem Server ein Laufzeit-Interpreter vorhanden sein, der das Programm ausführt. Wenn der Server zum Beispiel ein Unix-Rechner ist, führt er C-Programme aus, die mit einem Unix-C-Compiler zu einer ausführbaren Datei kompiliert wurden. Wenn der Server ein Windows-NT-Rechner ist, können CGI-Scripts auch EXE-Dateien sein, die mit 32-Bit-Compilern für C, Pascal, Visual Basic usw. erzeugt wurden. Die meisten heutigen CGI-Programme sind in der Unix-Shell-Sprache oder in Perl geschrieben. Die Unix-Shell-Sprache wird von allen Unix-Rechnern interpretiert. Für Perl muß ein entsprechender Interpreter installiert sein.

[107] Aus Sicht des Mieters von Speicherplatz auf einem WWW-Server steht die CGI-Schnittstelle in Form eines bestimmten Verzeichnisses zur Verfügung. Meistens hat dieses Verzeichnis den Namen cgi-bin. In diesem Verzeichnis können Programme abgelegt werden, die CGI-Aufgaben übernehmen.



In dem Beispiel kann der Anwender in einer angezeigten HTML-Datei i (Formular) Daten eingeben, zum Beispiel eine Suche in einer Datenbank formulieren. Nach dem Abschicken des Formulars an den Server-Rechner wird ein CGI-Programm aufgerufen. Das CGI-Programm setzt die vom Anwender eingegebenen Daten in eine Datenbankabfrage um. Die Datenbankanwendung liefert die Suchergebnisse an das aufrufende CGI-Programm zurück (oder schreibt sie in eine Datei, die dasCGI-Programm dann auslesen kann). Das CGI-Programm erzeugt nun HTML-Code, wobei es die Suchergebnisse als Daten in den HTML-Code einbaut. Den HTML-Code sendet das CGI-Programm an den WWW-Browser, der die Suchabfrage gestartet hat. Am Bildschirm des Anwenders verschwindet die WWW-Seite mit dem Suchformular. Stattdessen erscheint eine neue Seite mit den Suchergebnissen, dynamisch generiert von dem CGI-Programm.

Abb.: CGI-Situation für Suchdienste im WWW

PHP ist eine Skriptsprache, die direkt in HTML-Seiten eingebettet wird, d.h. der Autor schreibt PHP-Befehle zusammen mit HTML-Befehlen in eine Datei. Wird diese Datei von einem Betrachter angefordert, so werden diese PHP-Befehle von einer "Zusatzsoftware" des Webservers[108] Schritt für Schritt ausgeführt und die Ergebnisse an den Betrachter weitergeleitet..
PHP wird seit etwa 1994 entwickelt und erfreut sich stetig wachsender Beliebtheit. Ein besonderer Schwerpunkt liegt auf der Einbindung verschiedener Datenbanken. Die Sprache ist an C, Java und Perl angelehnt.
Prinzipiell kann PHP alles, was jedes andere CGI-Programm kann, also bspw. Formulardaten sammeln, dynamischen Inhalt von Websites generieren, Cookies senden oder empfangen. Die größte und bemerkenswerteste Stärke von PHP ist seine Unterstützung für zahlreiche Datenbanken.
Bei der **verteilten Datenverwaltung** werden die Aufgaben zwischen Client und Server aufgeteilt. Beispiele sind Entwicklungsumgebungen, die sowohl auf lokale als auch entfernte Datenbanken zugreifen können. Die verteilte Datenverwaltung geht tief in Theorie und Konzept verteilter Datenbanken ein.
Bei der **kooperativen Verarbeitung** findet die Datenhaltung Server-seitig statt. Repräsentationsaufgaben werden vom Client wahrgenommen und die Applikations-Funktionsschicht wird zwischen Server und Client aufgeteilt.

[108] Der Webserver muß "PHP"-fähig sein. Je nach Installation interpretiert diese PHP-Zusatzsoftware nur Dateien mit der Endung ".php3"

Bei der **entfernten Verarbeitung** wird die Datenhaltung vom Server wahrgenommen. Die Verarbeitung findet „Client“-lastig statt, d.h. der Client übernimmt die Anwendungs- und Repräsentationsaufgaben. Die Anwendungslogik kann somit vollkommen auf der Seite vom Browser, d.h. vom Client ausgeführt werden. Ein solches Programm, das innerhalb vom Browser ausgeführt wird, ist ein Applet. Eine besondere Rolle spielt hier die Programmiersprache Java. In Java geschriebene Programme können in einen maschinenunabhängigen Zwischencode übersetzt und Server-seitig abgelegt werden. Wenn eine HTML-Seite auf ein Java-Programm, ein sog. Applet verweist, dann wird es zum Client übertragen und dort mit einem entsprechenden Interpreter ausgeführt. So steht nichts mehr im Wege, auf dem Client Animationen ablaufen zu lassen, Eingabefelder zu überprüfen oder auch Datenbank-Anweisungen (SQL-) aufzurufen.
Einen geeigneten Satz von Funktionen für den Datenbankzugriff unter Java enthält die JDBC (Java Database Connectivity)-Schnittstelle. JDBC ist die Spezifikation einer Schnittstelle zwischen Client-Anwendung und SQL.
Die Bearbeitung von Daten einer Datenbank durch ein Java-Programm umfaßt im Rahmen der JDBC-Schnittstelle:

Zuerst ruft das Java-Programm die getConnection()-Methode auf, um das Connection-Objekt zu erstellen. Dann ergänzt es das Statement-Objekt und bereitet die SQL-Anweisungen vor. Ein SQL-Statement kann sofort ausgeführt werden (Statement-Objekt) oder ein kompilierbares Statement (Prepared-Statement-Objekt) oder ein Aufruf an eine gespeicherte Prozedur (Callable-Statement-Objekt) sein. Falls die executeQuery()-Methode ausgeführt wird, wird ein Resultat-Objekt zurückgeschickt. SQL-Anweisungen wie update und delete verwenden die executeUpdate()-Methode. Sie gibt einen ganzzahligen Wert zurück, der die Anzahl der Zeilen anzeigt, die vom SQL-Statement betroffen sind.
Der ResultSet enthält Zeilen mit Daten, die mit der Methode next() abgearbeitet werden können. Bei einer Anwendung mit Transaktionsverarbeitung können Methoden wie rollback() und commit() verwendet werden.

Zur Ausführung kann ein Java-Programm über mehrere JDBC-Treiber angesteuert werden. Jeder Treiber registriert sich beim JDBC-Manager währen der Initialisierung. Beim Versuch, sich mit einer Datenbank zu verbinden, gibt das Java-Programm einen Datenbank URL an den JDBC-Manager weiter. Der JDBC-Manager ruft dann einen der geladenen JDBC-Treiber auf. Die URLs von JDBC haben die Form "jdbc:subprotocol:subname". "subprotocol" ist der Name der jeweiligen Datenbank, z.B.: "jdbc:oracle:oci7:@rfhs8012_ora3"

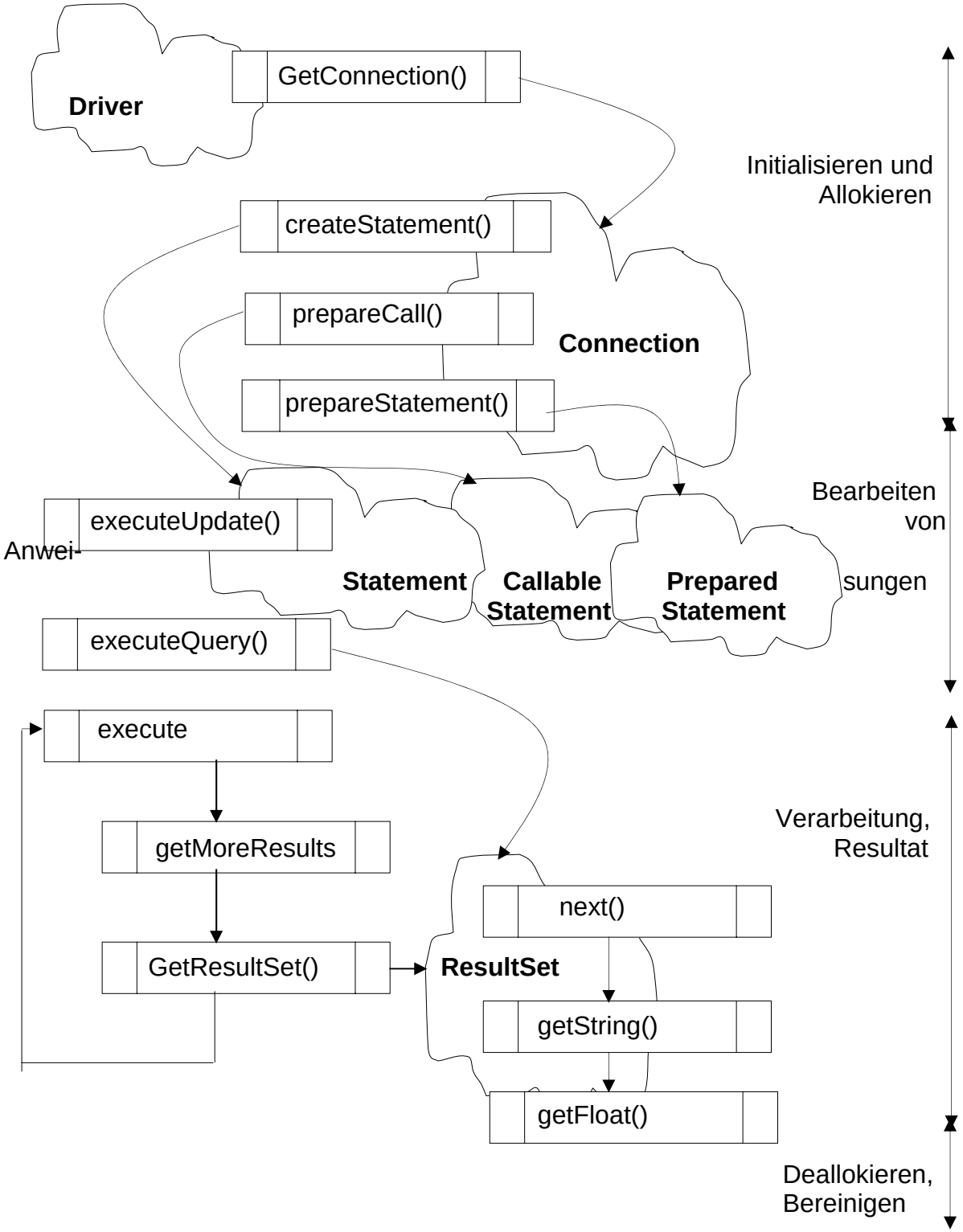


Abb. 1.5-20: JDBC-Objekte, Methoden, Ablauf

1.5.3.4 Web-Services

Unter Web-Services versteht man lose gekoppelte, verteilte Dienste, die über Internet-basierte Protokolee und XML-Nachrichten in einer sevice-orientierten Architektur veröffentlicht, lokalisiert und dynamisch aufgerufen werden können. Es existieren mehrere verschiedene, auf XML basierende Standards:

SOAP (simple object acces protocol) für den Dienstaufruf
WSDL (web service description language) zur Dienstbeschreibung
UDDI (universal description, discovery and integration) als Verzeichnisdienst zum Ankündigen und Auffinden von Diensten
BPEL4WS (busisness process execution language for web services)

Web-Services basieren auf Nachrichten, die in Form von XML-Dokumenten zwischen Server und Clients ausgetauscht werden. Diese Nachrichten werden über Internet-Protokolle wie bspw. HTTP oder E-Mail übertragen. Das Nachrichetenformat wird durch die SOAP-Standard festgelegt, der durch das W3C definiert wird. Web-Services können über SOAP Nachrichten aufrufen und mit Hilfe von WSDL auf eine einheitliche Art und Weise beschrieben werden. Die Beschreibungen werden durch den Verzeichnisdienst UDDI veröffentlicht. Zusätzlich können mit Hilfe von BPEL4WS komplexe Abfolgen von Web-Services als Geschäftsprozesse zusammengefaßt werden, als komplexer Dienst beschrieben und wiederum über WSDL veröffentlicht werden.

Prozeß- und Kompositionsschicht	BPEL4WS
Veröffentlichungsschicht	UDDI
Beschreibungsschicht	WSDL
Meldungsschicht	SOAP
Übertagunsschicht	HTTP

Abb.: Schichtenmodell für Web-Services

Web-Services können dann so realisiert werden: Ein Dienstanbieter publiziert eine Beschreibung seines Dienstes über WSDL im Verzeichnisdienst UDDI. Ein Dienstnachfrager sucht passende Dienste und deren technische Beschreibung in diesem Verzeichnis. Über den Verzeichnisdienst kann er diese finden und danach beim Dienstanbieter automatisiert aufrufen. Der Nachrichtenverkehr zwischen allen drei Partnern wird über SOAP-Nachrichten abgewickelt.

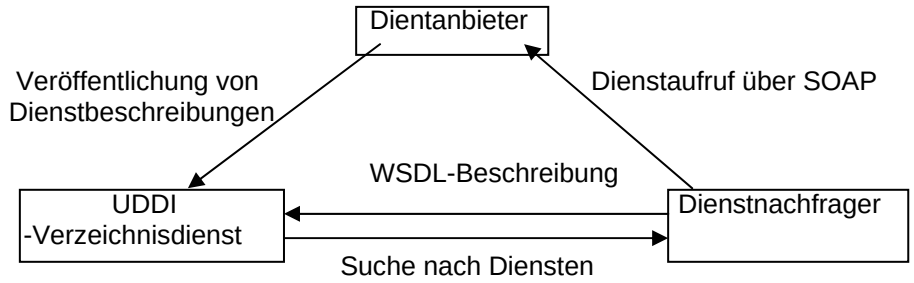


Abb.: Service-orientierte Architektur mit SOAP, WSDL und UDDI

Bedeutende Anwendungsbereiche von Web-Services sind u.a. Anwendungsintegration, elektronischer Datenaustausch, Electronic Business Anwendungen und Business-to-Business-Integration.

1.5.3.5 Oracle Application Server

Das Entwickeln von Applikationen für das Web ist ein komplexer Prozeß. Die Zahl der Programmiersprachen, APIs und Entwicklungsformen nimmt stetig zu. Oracle bietet eine Produktserie zur Entwicklung und zum Test von Anwendungen, zur Verbindung mit einer Oracle-DB und zur Integration mit minimalem Aufwand in das Web an:

Communication Services	Business Logic Services	Presentation Services	Business Intelligence Services
Oracle HTTP Server (Apache)[109] mod_plsl mod_jserv mod_oc4j mod_ose mod_dms mod_onsint mod_oprocmgr mod_oradav mod_ossl mod_osso	Oracle BC4/EJB Oracle 8i/9i/10g JVM Oracle 8/9i/10g PL/SQL Oracle Forms 10g Content Management Services Content Management SDK	Apache JServ[110] Oracle OC4J[111] Oracle JSP/PSP Portal Services Oracle Portal 10g	Oracle Reports 10g Oracle Discover Plus/viewer Caching Services Oracle Web Cache 10g
Developer's Toolkit			
XML Toolkit			
Content Management Toolkit			
Wireless Toolkit			
Portal development Toolkit			
System Services			
Oracle Enterprise Manager		Oracle Identity Management	

Communication Services: behandelt Anforderungen vom und ins Web
Business Logic Services: Entwicklungswerkzeuge und –sprachen für den Aufbau von Applikationen
Presentation Services: Entwicklungswerkzeuge und –sprchen für den Aufbau von Applikationen
Caching Services: Tools zur Verbesserung der Website Performance
Content Management Services: Tools zur Verwaltung von Dokumenten in der Datenbank
Portal Services: bietet Publikationsfunktionen für Inhalte und Portlets
Business Intelligence Services: bieten Reports und Ad-hoc Abfragen
Database Services: Die Oracle-datenbank zum Speichern von Applikationsdaten
Persistent Layer Services: Bietet ein objektrelationales Gerüst
Developer's Toolkits: Application Programming Interfaces (APIs) zur Unterstützung beim Anlegen von Applikationen

Abb.: Die Architektur von Oracle Application Server 10g

[109] Ab Oracle 9iAS Release 1 nutzt Oracle Apache als Web Server, der in der Oracle Dokumentation HTTP-Server heißt.
Weitere Informationen über Apache: http://www.apache.org
[110] Apache Jserv (mod_jserv) ist ein Modul für den Apache Web Server, der Sun's Java Servlet API zum Ausführen von serverseitigem Java-Code implementiert.
[111] OC4J ist die zentrale J2EE-Laufzeitkomponente von Oracle Application Server 10g

1.6 Verteilte Systeme

1.6.1 Verteilte Datenbanken

Definition und Anforderungen

"**Verteilte Datenbanken**" sind auf verschiedenen Rechnern eines Rechnernetzes verteilt und besitzen ein Datenbankverwaltungssystem (DBMS), das die Daten in einheitlicher Form verwaltet. Für den Benutzer eines derartigen Systems soll die physische Verteilung der Daten nicht sichtbar sein, d.h.: Das System der verteilten Datenbanken (VDBS) realisiert eine logisch zentrale Sicht der physisch dezentral abgespeicherten Daten. Die Information über die Datenverteilung wird im Systemkatalog gehalten.
Verteilte Datenbanksysteme (VDBS) lassen sich einteilen in homogene und heterogene Systeme. Ein VDBS ist homogen, falls Datenmodell und Software auf allen Rechnern einheitlich sind, andernfalls heterogen.
Verteilte Datenbanken werden in vielen Fällen der Organisationsstruktur eines Unternehmens besser gerecht als herkömmlich zentralisierte Datenspeicher, da Änderungen in der Datenverteilung keine Auswirkungen auf bestehende Anwendungen haben.
Die Verteilung auf mehrere DBMS (in der Regel auf verschiedenen, geographisch verteilten Rechnern) unterstützt auf der einen Seite das Lokalitätsverhalten vieler Anwendungen, ermöglicht andererseits auch die zentrale, logische Sicht der Daten.
Die wichtigsten Anforderungen, die Verteilte Datenbanksysteme „idealerweise“ erfüllen sollten, formulierte C.J. Date[112] in 12 „Regeln“:

1. Lokale Autonomie
Lokale Daten werden unabhängig von anderen Standorten verwaltet. Jeder Rechner sollte ein Maximum an Kontrolle über die auf ihm gespeicherten Daten ausüben. Der Zugriff auf diese Daten darf nicht von anderen Rechnern abhängen.
2. Keine zentralen Systemfunktionen
Zur Unterstützung einer hohen Autonomie der Knoten und Verfügbarkeit sollte die Datenbankverarbeitung nicht von zentralen Systemfunktionen abhängen. Solche Komponenten bilden außerdem einen potentiellen Leistungsengpaß.
3. Hohe Verfügbarkeit
Fehler im System (z.B. Rechnerausfall) oder Konfigurationsänderungen (Installation neuer Software und Hardware) unterbrechen nicht die Datenbankbearbeitung.
4. Ortstransparenz
Der physische Speicherort sollte für Benutzer verborgen bleiben. Der Datenbankzugriff darf sich vom Zugriff auf lokale Objekte nicht unterscheiden.
5. Fragmentierungstransparenz
Eine Relation (Tabelle) der Datenbank sollte verteilt auf mehrere Knoten gespeichert werden können. Die dazu zugrundeliegende (horizontale oder vertikale) Fragmentierung der Relation bleibt für den Datenbankbenutzer unsichtbar.
6. Replikationstransparenz
Die mehrfache Speicherung von Teilen der Datenbank auf unterschiedlichen Rechnern bleibt für den Benutzer unsichtbar. Die Wartung der Redundanz obliegt ausschließlich der DB-Software.
7. Verteilte Anfragebearbeitung

[112] C.J. Date: An Introduction to Database Systems, 5th Edition (chapter 23), Addison Wesley, 1990

Innerhalb einer DB-Operation (SQL-Anweisung) sollte die Möglichkeit bestehen, auf Daten mehrerer Rechner zuzugreifen. Zur effizienten Bearbeitung sind durch verteilte DBMS geeignete Techniken bereitzustellen (z.B. Query Optimierung)
8. Verteilte Transaktionsverarbeitung
Das DBMS hat die Transaktionseigenschaften auch bei der verteilten Bearbeitung einzuhalten. Dazu sollten geeignete Recovery- und Synchronisationstechniken bereitstehen.
9. Hardwareunabhängigkeit
Die Verarbeitung der Datenbank sollte auf verschiedenen Hardware-Plattformen möglich sein. Sämtliche Hardware-Eigenschaften bleiben dem Benutzer verborgen.
10. Betriebssystemunabhängigkeit
11. Netzwerkunabhängigkeit
Die verwendeten Kommunikationsprotokolle und -netzwerke haben keinen Einfluß auf die DB-Bearbeitung.
12. Datenbanksystemunabhängigkeit
Es muß möglich sein, unterschiedliche Datenbanksysteme auf den enzelnen Rechnern einzusetzen, solange sie eine einheitliche Benutzerschnittstelle (z.B. gemeinsame SQL-Version) anbieten.

Referenzarchitektur für verteilte Datenbanken

Sie beschreibt ein idealtypisches Modell[113] für konzeptionelle Schichten in verteilten Datenbanken:

Globales Schema
Es beschreibt Datenobjekte, ihre Integritätsbedingungen und Berechnungs-strukturen aus globaler, d.h. nicht verteilter Sicht.
Fragmentierungsschema
Es teilt eine globale Relation (Tabelle) in ein oder mehrere logische Teile (Fragmente) und verteilt diese auf geeignete Rechnerknoten. Fragmente des Fragmetierungsschemas müssen bzgl. Auf die ihnen zugeordneten globalen Relationen drei Bedingungen erfüllen:

1. Alle Daten der globalen Relation müssen in ihren Fragmenten vertreten sein
2. Jede globale Relation muß sich aus ihren Fragmenten vollständig rekonstuieren lassen.
3. Die Fragmente sollen überschneidungsfrei sein[114].

Allokierungsschema
Es beschreibt, auf welchen Datenbankknoten des verteilten Systems ein Fragment physikalisch gespeichert wird (`1:1` oder `1:n` Verhältnis). Im Falle der `1:1`-Verteilung spricht man von einer Dispersion, andernfalls von einer Replikation der Daten. Die Replikation erlaubt die permanente Abbildung des Fragments.
Lokales Schema
Hier werden die physikalischen Abbilder in die Objekte und das Datenmodell des jeweiligen datenbanksystems umgesetzt. Das lokale Schema ist abhängig vom Typ des zugrundeliegenden Datenbanksystems. Die ersten drei Schichten sind unabhängig vom benutzten DBS.

[113] vgl.: Ceri, Stefano u. Pelagotti, Guiseppe: Distributed Databases, Principles & Systems, McGraw-Hill 1985
[114] Ausnahme: Bei vertikale Fragmentierung wird der Primärschlüssel redundant gespeichert, um die Rekonstruktion der globalen Relation zu gewährleisten.

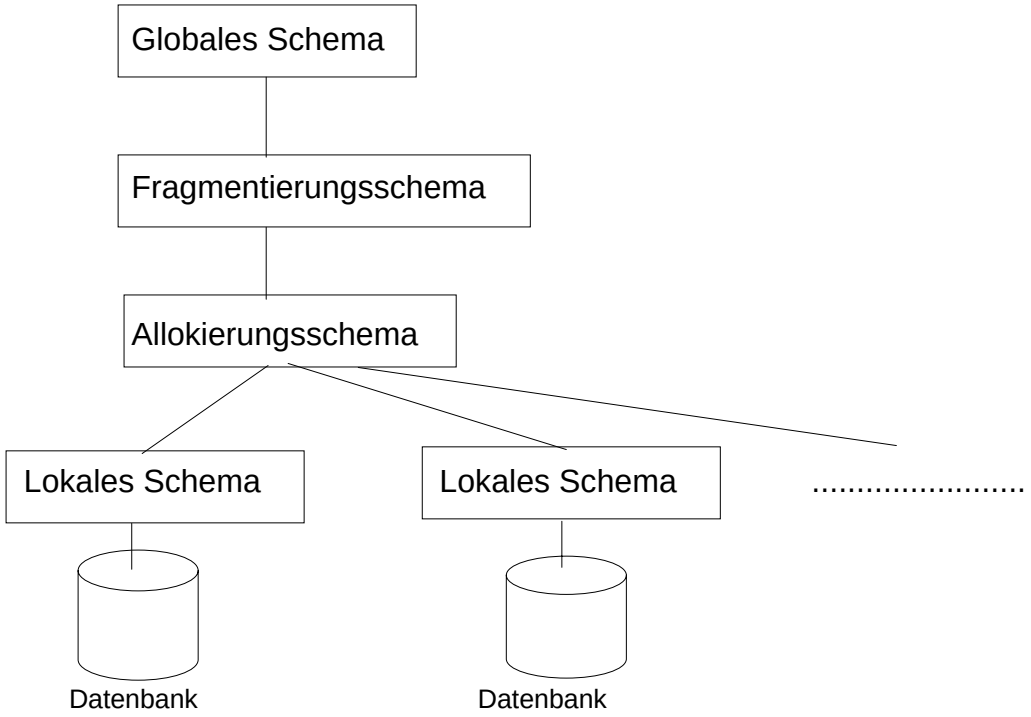


Abb. 1.5-21: Aufbau einer verteilten Datenbank

Rahmenbedingunen für das Design

Es gilt nicht nur wie bei zentralen Datenbanken das konzeptuelle Schema aufzubauen, es müssen auch Fragen der Fragmentierung, Allokierung bzw. Ortsverteilung gekärt werden. Wichtige Prinzipien für Verteilung und Replikation der Daten sind:

- Lokalitätsprinzip
  Daten sollen so nahe wie möglich an den Ort der Verarbeitung gelegt werden (Minimierung der in der Regel aufwendigen Zugriffe über das Netz).
- Optimierung der Verfügbarkeit und Zuverlässigkeit
  Durch geeignete Replikationen lassen sich die erhöhten Risiken komplexer Konfigurationen und Verwltung verteilter Datenbanken minimieren. Im Falle eines Systemabsturzes oder eines Netzausfalls kann die betroffene Applikation dann auf einen anderen Knoten mit replizierten Daten ausweichen.
- Optimierung der Lastverteilung durch Verteilung der Daten und der auf daten zugreifenden Applikationen unter Wahrung des Lokalitätsprinzips und größtmöglicher Parallelisierung der Verarbeitung.

Systemarchitektur

Physikalisch gesehen ist eine verteilte Datenbank ein über ein Telekommunikationssystem verbundenes Netz aus autonomen Rechnerknoten. Jeder Knoten verfügt über ein eigenes Datenbanksystem:
An die Datenbanksysteme der Rechnerknoten müssen sich sowohl lokale Abfragen ohne Berücksichtigung anderer Knoten absetzen und bearbeiten lassen als auch globale, mehrere Knoten einbeziehende Abfragen. Für diese Abfragen gelten die Forderungen nach Verteilungs-, Replikations- und

Fragmentierungestransparenz[115], d.h. jeder Rechnerknoten benötigt eine zusätzliche (globale) Komponete des **DBMS**, die diese Anforderungen erfüllt. Diese Komponente hat dann folgende Aufgaben:

- Bereitstellen von Wissen über die Verteilung der Daten (Speicherungsort, Replizierung, Fragmentierung mit netzweit eindeutiger Kennung). Die Bereitstellung dieser Metadaten übernimmt ein globales Datenwörterbuch.
- Erbringen der **ACID**-Eigenschaften im Rahmen einer globalen Transaktions-verwaltung, so daß ein Anwendungsprogramm glaubt, die Daten würden im **DBS** des Rechners gehalten, auf dem es abläuft.
- Lenkung der Zugriffe an die zuständigen Knoten im Rahmen der Auftrags-bearbeitung. Globale Anfragen werden in eine Reihe lokaler Anfragen aufgespalten, die an die betreffenden Knoten versandt werden und deren Antworten dann wieder eingesammelt und verknüpft werden. Die Auftragsbearbeitung erstellt dazu einen möglichst kostengünstigen Ausführungsplan (unter Ausnutzung der Kenntnisse von Speicherungsort, Replizierung, Fragmentierung und Übertragungskosten).

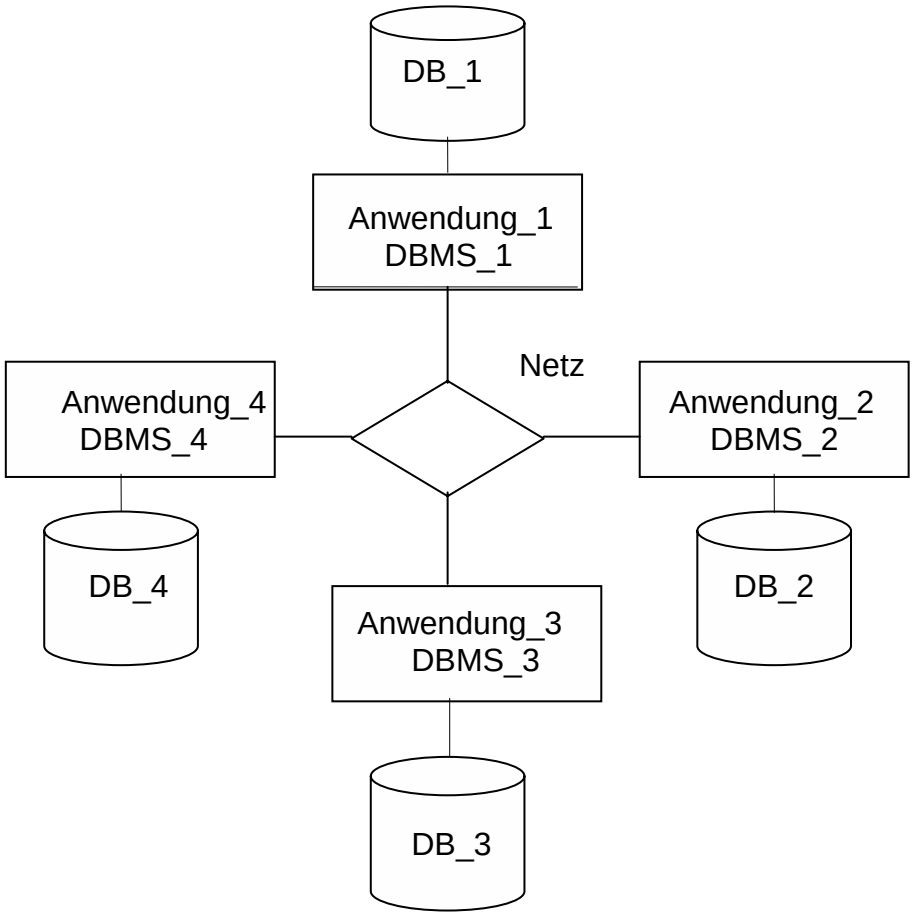


Abb. 1.5-22: Verteilte Datenbanken

Verteilungsstrategien

Verteilte Datenbanken können Verteilungen hinsichtlich der Daten, des Datenbankschemas, des Systemkatalogs, der Query-Auswertungen, der Synchronisation und der Verfahren zur Fehlerbehandlung auf verschiedene Arten

[115] Die Anwendung arbeitet mit globalen Relationen, d.h. der Programmierer braucht keinen Kenntnis über Fragmentierung und Allokation.

vornehmen. Das führt zu einer Fülle komplexer Probleme, die ein "Verteiltes Datenbanksystem" lösen muß.

Datenverteilung (Fragmentierung)

Die Fragmentierung ermöglicht es, Daten einer globalen Relation (Tabelle) auf unterschiedliche Knoten des verteilten Systems zu speichern. Abhängig von der Art, wie Tupel einer globalen Relation aufgeteilt werden, spricht man von horizontaler oder vertikaler Fragmentierung bzw. von einer Mischform der beiden.
Unter **horizontaler Fragmentierung** werden komplexe Tupel einer globalen Relation mit Hilfe der Selektion aufgeteilt. Die Relation (Tabelle) wird horizontal geschnitten.
Bsp.: Horizontale Fragmentierung der Relation [116] "Konto" nach Filialen. Jede Filiale speichert auf ihrem Rechner die sie betreffenden Daten.

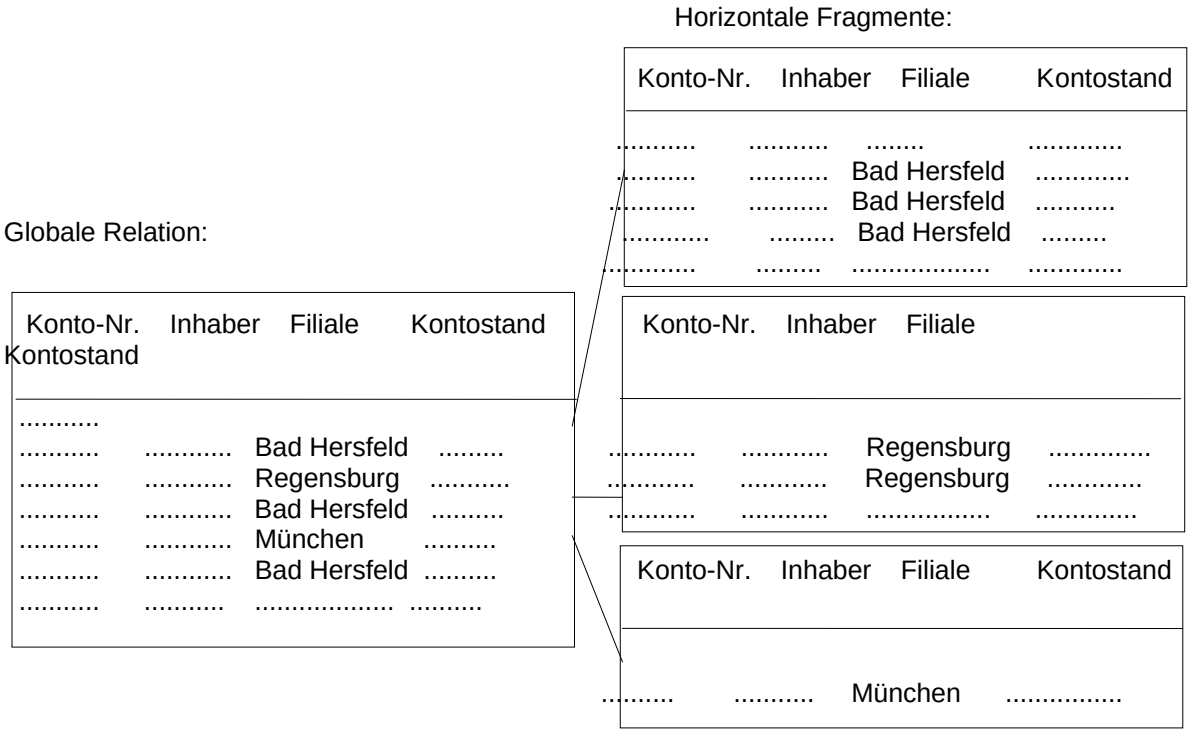


Abb. 1.5-23: Horizontale Fragmentierung der Kontorelation

Die **vertikale Fragmentierung** teilt die globale Relation in Abhängigkeit von den Attributen auf. Die Fragmentierung wird mit Hilfe der Projektion gebildet. Die Zuordnung der verteilt gespeicherten Attributgruppen erfolgt über den redundant abgebildeten Primärschlüssel.
Bsp.: Aufteilung der Relation "Artikel" in ein bestandsbezogenes und in ein preisbezogenes Tupel.

[116] In relationalen Datenbanken ist die Relation (Tabelle) Ausgangspunkt für die Verteilungseinheit

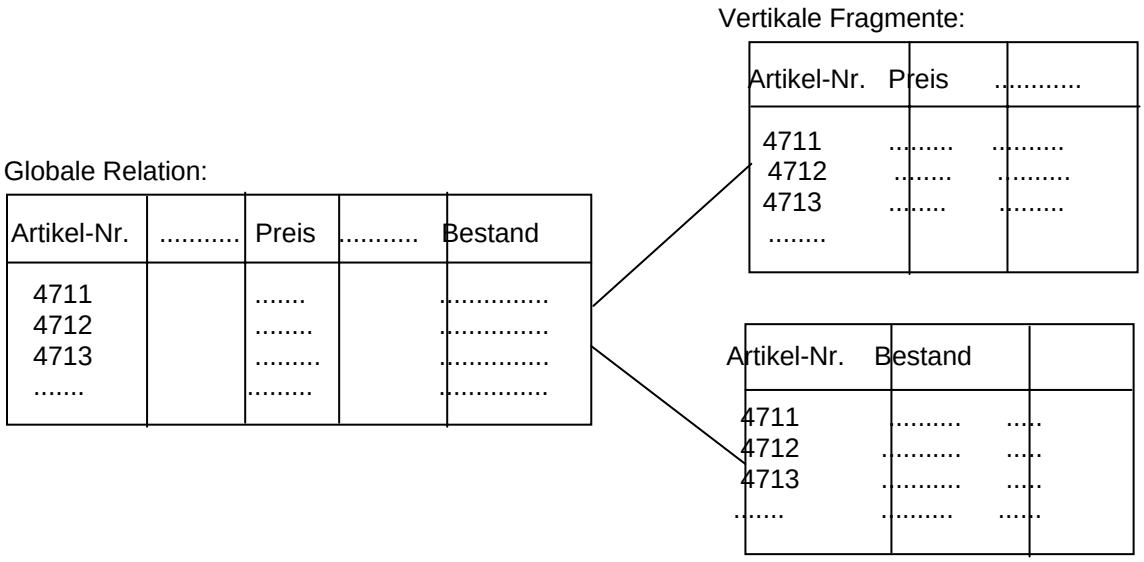


Abb. 1.5-24: Vertikale Fragmentierung der Realtion Artikel

Auch Mischformen sind denkbar. Dabei läßt sich bspw. ein vertikal gebildetes Fragment nochmals in eine Gruppe von horizontal gebildeten Fragmenten unterteilen.
Auf dem Fragmetierungsschema baut das Allakationsschema auf. Hier ist eine Entscheidung über Dispersion und Replikation der Fragmente zu treffen. Bei der Dispersion werden das Fragment einer globalen Relation genau einmal auf einen Knoten des verteilten Systems als Tabelle abgebildet. Der Zugriff auf die im Fragment gespeicherten Daten hat auf genau diesen Knoten zu erfolgen und ist damit abhängig von der Verfügbarkeit des betreffenden Knotens und der Verfügbarkeit des Netzwerks zur Erreichung des Knotens. Bei der Replikation werden Fragmente redundant gespeichert. Der Verbrauch an Speicherplatz steigt, auf die Daten kann aus mehreren Lokalitäten "näher" und "performanter" zugegriffen werden.

Katalogorganisation

Bei einem nicht voll redundanten VDBS muß zur Beantwortung der Anfrage (query processing) bestimmt werden, an welchem Knoten die angesprochenen Daten gespeichert sind. Informationen über Heimatadressen befinden sich im Systemkatalog (data dictionary, data directory). Für die Abspeicherung und Verwaltung der Katalogdaten sind ähnliche Probleme zu lösen wie bei den Benutzerdaten. Wichtige Formen der Katalogorganisation sind:

- zentralisierter Katalog
  Ein Rechner hält den vollständigen Katalog. Das Gesamtsystem ist allerdings von Leisung und Ausfallsicherheit eines Rechners abhängig
- Voll redundanter Katalog
  Jeder Rechner im verteilten System hält den Gesamtkatalog
- Lokaler Katalog
  Jeder Rechner hält einen Katalog über die lokale Dateien. Für verteilte Transaktionen entstehen Übertragungskosten, da die Adresse von Fremddateien nicht bekannt sind.

Schemaverteilung

Eine Datenbank wird bekanntlich durch das Datenbankschema beschrieben. Dort erfolgt die Definition

- der logischen Datenstrukturen, z.B. Relationen mit zugehörigen Attributen und Beziehungen zwischen den Relationen
- der physikalischen Strukturen z.B. Speicherstrukturen, Zugriffspfade

Ein **VDBS** benötigt zusätzlich die Definition der Verteilungsregeln, die einmal die Einheiten der Datenverteilung festlegen und zum anderen die Zuordnung der Dateneinheiten zu den Rechnern an unterschiedlichen Orten.
Verteilungsstrategien zur Schemaverteilung können unter dem Aspekt "Zentralisierung" bzw. "Verteilung" betrachtet werden. Es bestehen folgende Möglichkeiten, das Schema in die Verteilung mit einzubeziehen:

1. Nicht redundante Strategien

Zentralisierung: Die vollständige Schemabeschreibung befindet sich in einem Knoten. Jeder Verkehr mit der Datenbank impliziert einen Zugriff auf diesen ausgezeichneten Knoten.
Verteilung: An jedem Ort des Rechnerverbunds sind neben den lokalen Daten (deren Verteilung soll ebenfalls nicht redundant gespeichert sein) auch die zugehörigen Teile des Schema gespeichert. Ausschließlich lokale Verarbeitung benötigt keinen Netzzugriff mehr. Bei jeder nicht lokalen Anfrage (Query) muß in allen Knoten geprüft werden, ob sie die gewünschten Daten bereitstellen können.

2. Redundante Strategien

Zentralisierung: Jeder Knoten enthält die Schemainformation für seine Daten, zusätzlich steht aber das gesamte Schema in einer Zentrale zur Verfügung. Nicht lokale Zugriffe wenden sich zunächst an die Zentrale, um dort den Ort der gewünschten Daten zu erfahren.
Verteilung: Jeder Rechner kennt das vollständige Schema. Änderungen des Schemas sind an allen Rechnern gleichzeitig vorzunehmen. Benutzeraufträge können am Ort ihrer Eingabe vollständig gestartet und verwaltet werden.
Kombinationsmöglichkeiten: Irgendeine Teilmenge des Schemas kann als Verteilungseinheit herangezogen werden. Das bringt große Flexibilität, erzwingt den Aufbau eines "Schemas für das Schema (directory directory) zur Festlegung, welche Teile wo gespeichert sind.

Query Processing

Zwei wesentliche Unterschiede kenn zeichnen die Bearbeitung von Datenbankabfragen (Queries) in einem **VDBS** gegenüber einem zentralen **DBS**:

1. Zusätzliche Verzögerungen durch Netzkommunikation
2. Nutzung von Parallelität durch Beteiligung mehrerer Rechner und redundant vorhandener Daten

Bekannt sind folgende "query processing"-Strategien:

- Zerlegen der Anfrage an zentraler Stelle so, daß jede Teilabfrage von genau einem Knoten dezentral bearbeitet werden kann. Das Formuliern der zu verschickenden Teilabfragen erfolgt in

der Abfragesprache des Datenbanksystems (DML), z.B. in SQL (Versenden von DML-Anfragen zwischen Programmen).
- Ausarbeiten eines Gesamtplans zur Bearbeitung der Anfrage
  Der Ausführungsplan umfaßt:
  -- Zerlegung der kompexen Anfragen in Teilfragen
  -- Festlegen der Ausführungsreihenfolge dieser Teilfragen (sequentiell, parallel,..)
  -- Bestimmen der Knoten an die Daten geschickt werden sollen
  -- Auswahl lokaler Zufriffspfade
  -- Optimierung
     Hier ist festzulegen, wie lokale Bearbeitung und Übertragung von Teilergebnissen aufeinander folgen bzw. sich abwechseln sollen

Zielsetzung einer Anfrageübersetzung ist es, zu einer Anfrage einen Ausführungsplan festzulegen, der eine möglichst effiziente Bearbeitung des Auftrags ermöglicht. Diese Aufgabe ist bereits für zentralisierte DBS komplex, da im allg. eine große Anzahl alternativer Ausführungsstrategien besteht. Noch komplexer ist die Bestimmung eines optimalen Ausführungsplans im verteilten Fall.

Synchronisation

Falls mehrere Transaktionen (Prozese) in DBS gleichzeitig diesselben Daten benutzen, werden Synchronisationsmaßnamen erforderlich. Die parallele (nebenläufige, "concurrent") Ausführung muß dasselbe Ergebnis erzielen wie irgendeine sequentielle Reihenfolge. In einem VDBS gibt es darüber hinaus zusätzliche Probleme, bedingt bspw. durch die

- Existenz mehrfacher Kopien
- mögliche Beteiligung mehrerer Rechner an der Bearbeitung einer Transaktion

Verfahren zur Fehlerbehandlung (crash recovery)

Aufgabe der Verfahren ist es, bei Störungen

- die Datenbankkonsistenz zu erhalten (bzw. wieder herzustellen)
- Die Serialisierbarkeit aller benachbarten Transaktionen zu gewährleisten
- die Auswirkungen auf die direkt betroffenen Transaktionen zu beschränken

Die Basis der Fehlerbehandlungsverfahren bilden lokale Sicherungsverfahren in den Knoten des VDBS. Derartige Knoten verfügen über ein solides, sicheres Hintergrundsystem, dessen Inhalt alle Störungen überlebt. Lokale Sicherheit bietet die Grundlage für die Implementierung im globalen Fehlerbehandlungsverfahren.
Eine wesentliche Voraussetzung für die Bewahrung der Konsistenz in "verteilten DB" ist: Die Erhaltung der Unteilbarkeit globaler Änderungstransaktionen.
Bsp.: Eine Transaktion T hat in den Knoten K_i .. K_m über Teiltransaktionen T_i ... T_m neue Versionen erzeugt. Diese Versionen müssen in allen oder in keinem Knoten gültig gemacht werden. Es muß sichergestellt werden, daß sich etwa aufgrund eines Knotenzusammenbruchs eine Teiltransaktion zurückgesetzt, eine andere jedoch richtig beendet wird. Innerhalb einer einzelnen Transaktion T_i von Knoten K_i hat die zugehörige lokale Fehlerbehandlungsroutine für die Konsistenz der Daten gesorgt.

1.6.2 Datenbankrechner und Datenbankmaschinen

Hohe Softwarekosten und immer preiswertere Hardware haben dazu geführt, das Konzept des Universalrechners zu überdenken. Man kann heute einen Trend zur Aufteilung veschiedener Systemkomponenten auf autarke Prozessoren beobachten. Für Datenbanken sind zwei Enwicklungsrichtungen festzustellen:

1. Entwicklung neuer Rechnerfamilien, die sowohl assoziatives (inhaltsorientiertes) als auch paralleles Arbeiten ermöglichen. Diese Rechner führen die Suche nach Antwortmengen nicht mehr über physische Adressen aus, sondern ermöglichen das Wiederauffinden der Daten inhaltsmäßig über Assoziativspeicher.
2. Auf konventionellen Rechnerarchitekturen beruhende autarke Prozessoren

Die Konsequenz aus der 1. Entwicklungsrichtung führt zu spezieller Such- und Verarbeitungshardware und diejenige aus der 2. Entwicklungsrichtung zu dem Konzept der Datenbankrechner. Beide Trends führen zur Datenbankmaschine.

Ein **Datenbankrechner** ist ein Rechensystem, das ausschließlich Datenbankaufgaben wahrnimmt. Die Kommunikation zu einem Verarbeitungsrechner muß gewährleistet sein.
Eine **Datenbankmaschine**[117] ist ein Datenbankrechner, der für die Durchführung seiner Datenbankaufgabe durch spezielle, diese Aufgabe unterstützende Hardware ausgerüstet ist. Neben einem intelligenten Kontroller, der eine schnelle Reduktion der Gesamtmenge auf eine komprimierte Zwischenergebnismenge durchführt, sollte eine Datenbankmaschine auch hardwaregestützte Operationen (z.B. zur Bildung von invertierten Listen, Berechnen von Hashschlüsseln bzw. Sortieren und Verknüpfen von Tupelmengen) umfassen. Diese Operationen können leistungsmäßig entweder durch hohe Parallelität von vielen relativ einfachen Hardwarebausteinen (z.B. Mikroprozessoren) oder von wenigen speziell entwickelten Hochleistungsbausteinen durchgeführt werden.

1.7 Integritätsbedingungen und Integritätsregeln

Konsistenz oder Integrität bedeutet die Korrektheit der in der Datenbank gespeicherten Daten. Dazu gehört die technische oder physische Integrität, d.h. richtiges Arbeiten des Datenbanksystems. Darüber hinaus sollen die Daten aber weitgehend den Abhängigkeiten und Bedingungen genügen, denen die durch die Daten dargestellten realen Objekte unterworfen sind ("logische Konsistenz").

[117] vgl.: Eberhard, L. und andere: Datenbankmaschinen - Überblick über den derzeitigen Stand der Entwicklung. Informatik-Spektrum, Heft 4, S. 31 - 39, (1981)

1.7.1 Integritätsbedingungen

Sie können auf verschiedene Weise klassifiziert werden.

Physische Integrität

Darunter versteht man neben der physischen Datenkonsistenz

- die Vollständigkeit der Zugriffspfade
- die Vollständigkeit der Speicherstrukturen
- die Vollständigkeit der Beschreibungsinformationen

Semantische Integritätsbedingungen

Ganz allgemein versteht man unter semantischer Integrität die Übereinstimmung von realen und gespeicherten Daten. Speziell könnte man hier unterscheiden:

- Bereichsintegrität
  Sie umfaßt den Erhalt der Attributwerte innerhalb der Relationen. Jedes Attribut einer Relation hat einen bestimmten Wertebereich.
- Intra-relationale Integrität
  Sie umfaßt die Korrektheit der Beziehungen zwischen den Attributen in einer Relation (z.B. funktionale Abhängigkeiten). Eine der wichtigsten Integritätsreglen ist in diesem Zusammenhang: Die Eindeutigkeit des Schlüssels. Schlüsseleindeutigkeit bedeutet: Kein neues Tupel kann eingetragen werden, falls Schlüsselwerte mit Werten des bestehenden Tupels in dieser Realtion übereinstimmen.
- Entitätsintegrität
  Sie verbietet Null-Werte im Feld des Primärschlüssels und garantiert, daß Sätze erst dann in die Datenbank aufgenommen werden, wenn das wichtigste Attribut (der Satzschlüssel) vorhanden ist.
- Beziehungsintegrität (referentielle Integrität)
  Sie sorgt in relationalen Datenbanken dafür, daß jedem Wert eines Fremdschlüssels ein entsprechender Primärschlüssel des gleichen Bereichs zur Verfügung steht.

Idealerweise speichern Datenbankverwaltungssysteme (**DBMS**), die Integritätskonzepte unterstützen, Integritätsbedingungen (z.B. zulässige Wertebereiche für Daten) im **Data Dictionary**.

Pragmatische Integritätsbedingungen

Dazu zählen gesetzliche und organisatorische Vorschriften.

Ablaufintegrität

Hierunter versteht man die Datenkonsistenz bei der Verwaltung des Mehrbenutzer-betriebes. Die Ablaufintegrität kann beeinträchtigt werden, wenn mehrere Benutzer zugleich auf eine Datenbank zugreifen.
Falls eine Verletzung der Integrität festgestellt wurde, so muß sie wiederhergestellt werden. Dies geschieht durch einen Wiederherstellungsprozeß, der von der Art des Fehlers abhängig ist.
So verfügt das Datenbankverwaltungssystem in der Regel über eine transaktionsorientierte Fehlerbehebung. Eine Transaktion ist bekanntlich eine Folge von Operationen, die alle insgesamt abgeschlossen sein müssen. Zu jeder Transaktion wird ein Protokolleintrag (Anfangszeitpunkt der Transaktion,

Transaktionsnummer, Transaktionstyp, Datenobjekt vor Veränderung (before image), Zustand des Datenobjekts nach Veränderung (after image), Fehlerstatus, Endzeitpunkt der Transaktion) erstellt. Im Fehlerfall besteht die Möglichkeit fehlerfreie von fehlerhaften Transaktionen aus dem Protokoll zu rekonstruieren. Fehlerhafte Transaktionen können dann (in umgekehrter Reihenfolge) zurückgesetzt werden.

1.7.2 Integritätsregeln

Sie gewährleisten die logische Korrektheit (Widerspruchsfreiheit) der Datenbank und sorgen damit für die Einhaltung der **Integritätsbedingungen**. Sie können in Datenbanksystemen (z.B. in Oracle V 7) als **Trigger** oder **Constraints** implementiert werden. Constraints beschreiben Integritätsregeln im Rahmen der Datendefinitionen bei der Erstellung des Datenbank-Schema. **Trigger** sind Programme, die komplexe Fälle, die nicht mit **Constraints** gelöst werden können, beschreiben. In beiden Fällen erfolgt die Ausführung einer Integritätsregel ereignisorientiert, d.h. Das Ändern, Einfügen oder Löschen eines Datenelements führt zur Ausführung einer Integritätsregel.

2. Relationale Datenbanken

2.1 Entwurf relationaler Datenbanken durch Normalisieren

2.1.1 Normalformen

2.1.1.1 Relationen in der 1. Normalform

1. Erste Normalform (ENF)

Eine **Relation** befindet sich in der **ersten Normalform** (ENF), wenn sie keine Wiederholungsgruppen und nur Felder, die sich nicht weiter untergliedern lassen, enthält.

Zugelassene Operationen

- Ein Schlüssel, der auch als Fremdschlüssel benutzt wird, darf nicht isoliert modifiziert werden
- Nur ganze Sätze können gelöscht oder eingefügt werden

2. Wie werden Entitätsattribute und Beziehungsattribute mit Hilfe von Relationen dargestellt?

Eine Relation ist eine zweidimensionale Tabelle. Jede Tabellenspalte enthält ausgewählte Werte einer **Domäne**, jede Tabellenzeile enthält Fakten einer **Entität**.

Bsp.: Die Beziehungen zwischen Studenten (repräsentiert durch den Entitätsschlüssel S# mit den Ausprägungen {S1, S2, S3}) und Dozenten (repräsentiert durch die Entitätsschlüssel D# mit den Ausprägungen {D1, D2}) sollen betrachtet werden. Jeder Student hat einen Namen (NAME) und ein Alter (ALTER) (= Entitätsatribute). Jeder Dozent lehrt eine Programmiersprache. Die Sprachkenntnisse der Studenten sind unterschiedlich. Die Studenten werden außerdem noch von Dozenten beurteilt (Beziehungsattribut). Allen Attributen sind für jede spezielle Ausprägung Werte aus Wertebereichen (Domänen) zugeordnet. Zur Beschreibung in einer relationalen Datenbank gibt es dann:

1) Relationen zum Festhalten von Entitätsbeziehungen

```
NAME(S#,NAME)        ALTER(S#,WERT)       DOZIERT(D#,FACH)
     S1   Karl            S1   20               D1   C
     S2   Vera            S2   35               D2   Pascal
     S3   Vera            S3   26

SPRACHKENNTNISSE(S#,SPRACHE,KRITERIUM)
```

```
S1  C         gut
S1  Prolog    mittel
S2  C         gut
S2  Pascal    schlecht
S3  C         gut
S3  Pascal    mittel
```

Am einfachsten wäre es, all diese Tabellen, so weit es möglich ist, zu einer Relation zusammen zu fassen:

```
STUDENT_E(S#,SPRACHE, KRITERIUM, NAME, ALTER)
          S1 C          gut      Karl  20
          S1 Prolog     mittel   Karl  20
          S2 C          gut      Vera  35
          S2 Pascal     schlecht Vera  35
          S3 C          gut      Vera  26
          S3 Pascal     mittel   Vera  26
```

Die vorliegende Zusammenfassung ist jedoch keine korrekte Darstellung des Sachverhalts in einer relationalen Datenbank. Sie ermöglicht realitätskonforme Sachverhalte zu verletzen, z.B.: S3 hat neben C, Pascal auch Prolog gelernt. Das bedeutet: Hinzufügen des Tupels (S3,Prolog,schlecht,Maria,28). Der Einschub enthält realitätswidrige Aussagen (Maria,28). Trotzdem ist er systemmäßig tolerierbar, weil ein bisher noch nicht existenter Schlüsselwert (S3,Prolog) eingebracht wird.

2) Relationen zum Festhalten einer Beziehungsmenge

```
BELEHRT(D#,S#)
        D1 S1
        D1 S3
        D2 S1
        D2 S2
        D2 S3
```

3) Relation zum Festhalten eines Beziehungsattributs

```
BEURTEILUNG(D#,S#,KRITERIUM)
            D1 S1   gut
            D1 S3   mittel
            D2 S1   gut
            D2 S2   schlecht
            D2 S3   gut
```

Aufgabe: In einer Relation soll festgehalten werden, welche Personen, welche Sprachen mit welchem Qualitätskriterium können.

1. Beschreibe dazu eine Relation, in der SPRACHE als Domäne an einem Entitätsattribut partizipiert.

```
PERSON(PE#,....)
SPRACHK(PE#,SPRACHE,QUALITAET)
```

2. Gib eine relationsmäßige Darstellung der Problemstellung an, bei der SPRACHE als Entitätsmenge an einer Beziehungsmenge partizipiert, die ihrerseits an einem Beziehungsattribut beteiligt ist!

PERSON(PE#,....)
SPRACHE(SPRACHE,....)
SPRACHK(PE#,SPRACHE,QUALITAET)

Daraus folgt:

- Werden Sprachen im Sinne eines Attributs behandelt, so kann eine bestimmte Sprache nur im Zusammenhang mit einer die Sprache sprechenden Person genannt werden.
- Werden Sprachen im Sinne von Entitäten aufgefaßt, so kann eine bestimmte Sprache auch eigenständig (d.h. ohne Nennung einer die Sprache sprechenden Person) behandelt werden.

Damit kommt dem Entitätsbegriff folgende Bedeutung zu: Mit der Definition einer Entitätsmenge wird beabsichtigt, die der Entitätsmenge angehörenden Entitäten eigenständig (d.h. ohne Nennung anderweitiger Begriffe) behandeln zu können.

2.1.1.2 Die zweite Normalform (ZNF)

Was bezweckt die Normalisierung?

- Eliminieren von Redundanz
- Eliminieren von Schwierigkeiten in Zusammenhang mit Speicheroperationen (Einschub, Löschen, Modifikation)
- Eindeutiges Festhalten realitätskonformer Sachverhalte, d.h.: Ermitteln von Relationen, die keine Möglichkeiten bieten realitätskonforme, funktionale Abhängigkeiten zu verletzen.

Bsp.: Funktionale Abhängigkeiten in der Relation Student_E

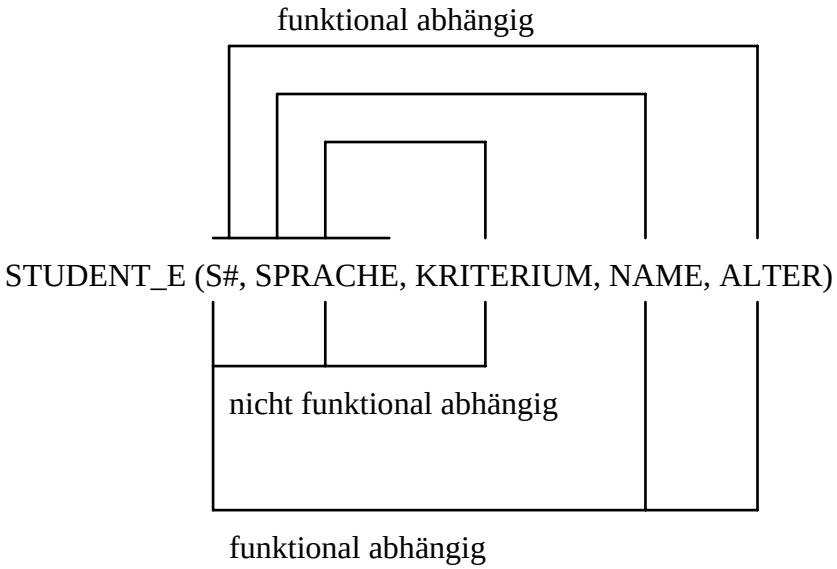


Abb. 2.1-1: Funktionale Abhängigkeiten in der Relation STUDENT_E

Nur die Spalte KRITERIUM ist nicht funktional abhängig vom Schlüsselteil S#. Alle übrigen Nichtschlüsselspalten sind vom Schlüsselteil S# funktional abhängig.

Normalisierungskriterium (**2. Normalform**)

In einer Relation sollte jede Nichtschlüsselspalte abhängig sein vom Gesamtschlüs-sel und nicht von Schlüsselteilen.

Überführung von der 1. in die 2. Normalform

Normalisierungskriterien verletzende Nichtschlüsselspalten werden aus der problembehafteten Relation eliminiert und zu neuen problemlosen Relationen vereinigt.

Bsp.:

STUDENT_E wird ersetzt durch

```
STUDENT(S#,NAME,ALTER)
        S1  Karl  20
        S2  Vera  35
        S3  Vera  26

SPRACHK(S#,SPRACHE,KRITERIUM)
        S1  C          gut
        S1  Prolog     mittel
        S2  C          gut
        S2  Pascal     schlecht
        S3  C          gut
        S3  Pascal     mittel
```

2.1.1.3 Die dritte Normalform (DNF)

Transitive Abhängigkeiten

Bsp.:

```
STUDENT_X(S#, NAME,  GEB  ,  ADR, FB#,   FBNAME   ,  DEKAN)
          S1  Karl  01.10.48 xxx  20  Informatik     Hechler
          S2  Vera  21.08.49 xxx  20  Informatik     Hechler
          S3  Vera  13.05.48 xxx  19  Elektrotechnik Jung
          S4  Maria 04.12.47 xxx  20  Informatik     Hechler
          S5  Tom   11.01.47 xxx  20  Informatik     Hechler
          S6  Fritz 01.03.49 xxx  19  Elektrotechnik Jung
```

Der Wechsel eines "Dekans" bewirkt hier mehrere Änderungen in der Tabelle (`update dependence`). Sie sind verursacht durch folgende Abhängigkeiten, die sich durch die folgenden Transitivitäten ausdrücken lassen:

$S\# \rightarrow FB\# \rightarrow DEKAN$

$S\# \rightarrow Dekan \rightarrow FBName$

STUDENT_X ist in der zweien Normalform (ZNF), nicht in dritter Normalform (DNF).

Dritte Normalform (DNF)

Eine Relation R ist in DNF, wenn sie sich in der ZNF befindet, und jedes Nichtschlüsselattribut von R nicht transitiv abhängig ist von jedem Schlüsselkandidaten.

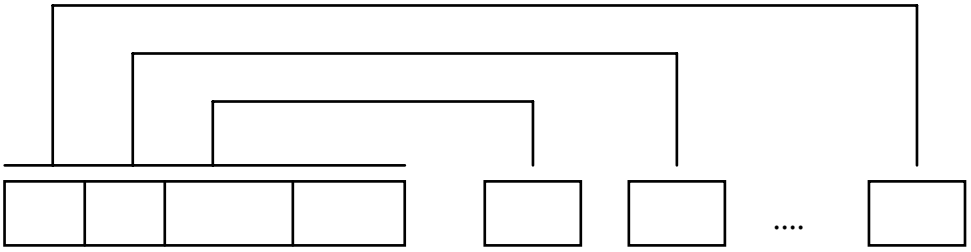


Abb. 2.1-2: Funktionale Abhängigkeiten in DNF

Überführung in die DNF

Bsp.:

```
STUDENT(S#, NAME, ADR,  GEB,   FB#)
        S1  Karl  xxx 01.10.48 20
        S2  Vera  xxx 21.08.48 20
        S3  Vera  xxx 13.05.48 19
        S4  Maria xxx 04.12.47 20
        S5  Tom   xxx 11.01.47 20
        S6  Fred  xxx 01.03.49 19

FACHBEREICH(FB#,   FBNAME   , DEKAN)
            20  Informatik     Hechler
            19  Elektrotechnik Jung
```

2.1.2 Spezielle Normalformen

Mit der DNF ist in der Regel der Normalisierungsprozeß abgeschlossen. Unter den folgenden Voraussetzungen ist aber der Normalisierungsprozeß nicht befriedigend beendet:

- die Relation hat mehrere Schlüsselkandidaten
- die Schlüsselkandidaten sind zusammengesetzt, bestehen also aus mehreren Attributen
- die Schlüsselkandidaten überlappen sich mit dem Primärschlüssel, d.h.: Sie haben mindestens ein Attribut mit dem Primärschlüssel gemeinsam

Falls Relationen mehrere zusammengesetzte und sich überlappende Schlüsselkandidaten aufweisen, wird die Boyce/Codd-Normalform zur Normalisierung herangezogen.

Boyce/Codd-Normalform (BCNF)

Eine Relation ist dann in BCNF, falls jedes (funktional) determinierendes Attribut zugleich Schlüsselkandidat ist.

Ein (funktional) determinierendes Attribut (z.B. A_i) bzw. eine determinierende Attributkombination liegt dann vor, wenn jeder Attributwert des determinierenden Attributs (bzw. der determinierenden Attributkombination) genau einen Attributwert eines anderen Attributs (z.B. A_j) bzw. einer anderen Attributkombination festlegt. Man spricht in diesem Zusammenhang auch von Determinanten. Eine Determinante ist demnach ein Attribut oder eine Gruppe von Attributen (Schlüsselkandidat), von der beliebig andere Attribute funktional abhängig sind.

Bsp.: Gegeben sind folgende Fakten

1. Ein Student belegt eine bestimmte Vorlesung bei einem Dozenten
2. Ein Dozent hält Vorlesungen nur zu einem Thema (d.h. lehrt nur ein Fach)
3. Ein Vorlesungsthema kann von mehreren Dozenten unterrichtet werden

Lösungsversuche:

1. BELEGUNG(DOZENT#, STUDENT#, VORLESUNG)

Das Schema dieser Relation zeigt eine teilweise Abhängigkeit (DOZENT# $\rightarrow$ VORLESUNG) und ist nicht einmal in 2. Normalform

2. BELEGUNG(STUDENT#, VORLESUNG, DOZENT#)

Die Relation ist in DNF, nicht aber in BCNF, da die Abhängigkeit "DOZENT# $\rightarrow$ VORLESUNG" besteht und "DOZENT#" ein Schlüsselkandidat ist. Die Lösung zeigt folgende Mängel:

a) Bevor nicht ein Student die Vorlesung belegt, kann kein Dozent, der die Vorlesung hält, eingetragen werden.
b) Für jeden Studenten, der eine bestimmte Vorlesung belegt hat, wird der Dozent, der die Vorlesung hält, redundant gespeichert.

Lösung: Sie kann durch Zerlegung der Relation in zwei Relationen erreicht werden. Die Attribute, die die BCNF verletzen, werden in eine abgetrennte Tabelle herausgezogen. Das determinierende Attribut übernimmt die Funktion des Primärschlüssels.

```
TRIFFT(DOZENT#, STUDENT#)
HAELT(DOZENT#, VORLESUNG)
```

Der Grund für die fehlerhafte Lösungsmöglichkeit (unter 1. und 2.) liegt in der falschen Darstellung der Beziehung DOZENT, STUDENT, VORLESUNG. Es liegt hier keine Dreifachbeziehung vor, sondern nur zwei unabhängige Zweierbeziehungen, wie es das folgende ERM-Diagramm zeigt:

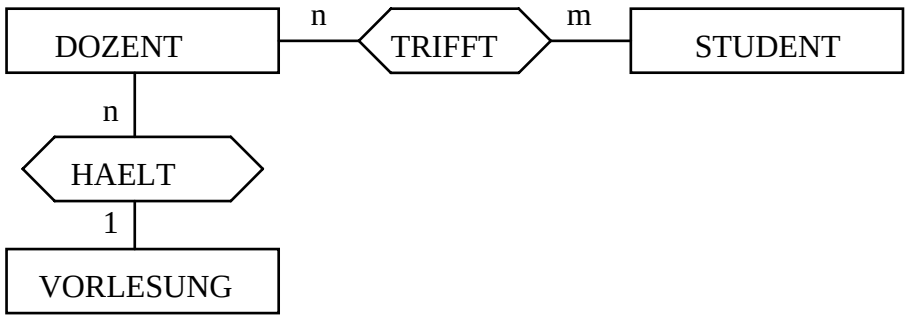


Abb. 2.1-3: ERM-Diagramm zur Beschreibung der Zweifachbeziehungen

Ziel eines Datenbankentwurfs ist ein Datenbankschema, dessen Relationenschemata möglichst in BCNF sind. Das kann durch fortschreitendes Zerlegen von Schemata , die diese Kriterien nicht erfüllen, erreicht werden.

Vierte Normalform (VNF)

Ein einführendes Beispiel: Gegeben ist die folgende, nicht normalisierte Relation:

VORLESUNG_DOZENT		
VORLESUNG	DOZENTEN_NAME	MERKMAL
DB (Datenbanken) AE (Anwendungsentwicklung)	Jürgen, Ernst Alex	Grundlagen, Übungen Einführung, Übungen

Abb. 2.1-4: Beispiel einer nicht normalisierten Relation

In dieser Tabelle werden folgende Fakten ausgedrückt:

- Eine bestimmte Vorlesung kann von beliebig vielen Dozenten gehalten werden und kann beliebig viele Merkmale zeigen.
- Zwischen Dozenten und Merkmalen eines Kurses (Vorlesung) bestehen keinerlei Abhängigkeiten
- Dozenten und Merkmale können mit jedem beliebigen Kurs in Verbindung stehen

Es handelt sich um eine Dreifachbeziehung, die normalisiert folgende Form aufweist:

VORLESUNG_DOZENT_NORMALISIERT		
VORLESUNG	DOZENT_NAME	MERKMAL
DB	Jürgen	Grundlagen
DB	Jürgen	Übungen
DB	Ernst	Grundlagen
DB	Ernst	Übungen
AE	Alex	Einführung
AE	Alex	Übungen

Abb. 2.1-5: Normalisierte Relation mit redundanten Daten

Die vorliegende, normalisierte Relation enthält redundante Daten, die zu Anomalien führen können!. Soll z.B. ein neuer Dozent Datenbanken (DB) mit den bekannten Merkmalen lehren, dann sind zwei Tabellenzeilen in die Relation aufzunehmen.
Die vorliegende, normalisierte Relation ist aber sogar in BCNF, denn alle Attributwerte sind Primärschlüssel. Es gibt außer der Kombination der Attribute VORLESUNG, DOZENT-NAME, MERKMAL kein weiteres funktional determinierendes Attribut.

Spaltet man die vorliegende Tabelle in zwei Relationen auf (Projektionen der ursprünglichen Relation), so ergibt sich die Lösung des Problems:

VORLESUNG_DOZENT_VNF	
VORLESUNG	DOZENT_NAME
DB	Jürgen
DB	Ernst
AE	Alex

VORLESUNG	
VORLESUNG	MERKMAL
DB	Grundlagen
DB	Übungen
AE	Einführung
AE	Übungen

Abb. 2.1-6: Relation in VNF

Für jede von einer bestimmten Person gehaltene Vorlesung gibt es eine identische Menge von Merkmalswerten. Man sagt, daß Attribut MERKMAL ist **mehrwertig abhängig** (multi value dependent) vom Attribut VORLESUNG. bzw. DOZENT.
Das führt zu der folgenden Definition einer mehrwertigen Abhängigkeit:

In dem Relationenschema $R = \{A_i, A_j, A_k\}$ ist das Attribut A_k vom Attribut A_i mehrwertig abhängig, falls zu einem A_i-Wert, für jede Kombination dieses A_i-Werts mit einem A_j-Wert, eine identische Menge von A_k-Werten erscheint.
Mehrwertige Abhängigkeiten erscheinen in einer Relation immer paarweise.
Werden diese mehrwertigen Abhängigkeiten beseitigt, dann spricht man von einer Relation in **vierter Normalform** (**VNF**).
Eine Relation ist in VNF, falls sie in DNF bzw. BCNF ist und keine mehrwertigen Abhängigkeiten zeigt.
Verstöße gegen die vierte Normalform entstehen häufig bereits in der Entwurfsphase. So hätte man im vorliegenden Beispiel erkennen können, daß die Beziehung zwischen VORLESUNG und DOZENT bzw. VORLESUNG und MERKMAL voneinander unabhängig sind und keine Dreifachbeziehung darstellen.

2.1.3 Der Normalisierungsprozeß

Zwei Methoden wurden für den Entwurf relationaler Datenbanken vorgeschlagen. Die erste Methode[118] ist ein Zerlegungsprozeß. Ausgangspunkt ist ein universelles Schema. Dieses Schema ist eine komplexe, häufig sogar verschachtelte Tabelle, die alle Attribute umfaßt. Die Regeln zur Normalisierung überführen diese Tabelle in einfache Tabellen.
Die zweite Methode[119] ist eine Synthese einer Menge funktionaler Abhängigkeiten. Bei der Zusammenfassung spielen die Regeln zur Normalisierung eine wichtige Rolle.

[118] vorgeschlagen von Codd, E.F. : "Normalized Data Base Structure: A Brief Tutorial", ACM SIGFIDET Workshop on Data Description, Access and Control, November 1971, Seiten 1 - 17
[119] vgl. Bernstein, P.: "Synthesizing Third Normal Form from Functional Dependencies", ACM Transactions on Data Base Systems, December 1976, Seiten 277 - 298

2.1.3.1 Die Normalisierung als Zerlegungsprozeß

1) Problemstellung

Fallbeispiel: Gesucht ist ein Datenmodell für eine relationale Datenbank, das die Entitätstypen PERSON (Mitarbeiter), ABTEILUNG, PRODUKT berücksichtigt.

Eine Analyse führt zu folgenden Fakten:

```
1. Fakt                    Ein Mitarbeiter hat einen Namen und
2. Fakt                    einen Wohnort. Er ist
3. Fakt                    in einer Abteilung tätig und arbeitet an
4. Fakt                    mehreren Produkten jeweils
5. Fakt                    eine vorbestimmte Zeit.
                           Jede Abteilung hat
6. Fakt                    einen Namen.
                           Jedem Produkt ist
7. Fakt                    ein Name zugeordnet. In einer Abteilung sind
8. Fakt                    mehrere Personen tätig. An einem Produkt
                           arbeiten in der
9. Fakt                    Regel mehrere Personen.
```

Erkennbar sind:

- Entitäten (PERSON, ABTEILUNG, PRODUKT)
- Beziehungen zwischen Entitäten (vgl. 3.,4.,8.,9. Fakt)
- Entitätsattribute (vgl. 1.,2.,6.,7. Fakt)
- ein Beziehungsattribut (vgl. 5 Fakt)

2) Nichtnormalisierte Relation

Die Fakten lassen sich alle in der folgenden Tabelle zusammen fassen:

```
PERSON_UN(PE#,NAME,WOHNORT,  A#, A-NAME,PR#,   PR-NAME, ZEIT)
          101 Hans Regensburg 1  Physik 11,12    A,B   60,40
          102 Rolf Nürnberg   2  Chemie 13       C     100
          103 Udo  München    2  Chemie 11,12,13 A,B,C 20,50,30
          104 Paul Regensburg 1  Physik 11,13    A,C   80,20
```

Nachteile dieser Zusammenfassung sind:

1. Die Anzahl der Elemente variiert von Zeile zu Zeile (schwierige Handhabung)
2. Mit nichtnormalisierten Relationen gibt es Schwierigkeiten bei Speicheropera-tionen (Verletzung realitätskonformer Sachverhalte)
3. Komplexe Verifikations-Programme sind erforderlich, die sicherstellen, daß ein und dasselbe, redundant auftretende Faktum auch nach der Speicheroperation durchgehend den gleichen Wert aufweist.

3) ENF

Die unter 2) angegebene Relation wird in die erste Normalform überführt. Hier ist in Verbindung mit einem bestimmten Schlüsselwert immer höchstens ein einziger Wert eines Attributs erlaubt.

```
PERSON_ENF(PE#,NAME,WOHNORT,   A#,A-NAME,PR#,PR-NAME,ZEIT)
           101 Hans Regensburg  1 Physik  11  A      60
           101 Hans Regensburg  1 Physik  12  B      40
           102 Rolf Nürnberg    2 Chemie  13  C      100
           103 Udo  München     2 Chemie  11  A      20
```

```
          103 Udo  München     2 Chemie  12  B      50
          103 Udo  München     2 Chemie  13  C      30
          104 Paul Regensburg  1 Physik  11  A      80
          104 Paul Regensburg  1 Physik  13  C      20
```

Ein PE#-Wert steht in der Regel mit mehreren PR#-Werten in Beziehung. Umgekehrt steht ein PR#-Wert mit mehreren PE#-Werten in Beziehung.
Schwierigkeiten mit Speicheroperationen (Speicheranomalien) sind immer noch vorhanden. Update- und Einschuboperationen ermöglichen (entgegen den genannten Fakten):

- Eine Person hat mehrere Namen, mehrere Wohnorte, mehrere Abteilungen.
- Für eine Abteilung und ein Produkt können mehrere Namen vergeben werden.

4) ZNF

Hier ist jedes nicht dem Schlüssel zugehörende Attribut funktional abhängig vom Gesamtschlüssel (nicht von den einzelnen Schlüsselteilen).
Die Analyse der ENF führt zu den folgenden funktionalen Abhängigkeiten:

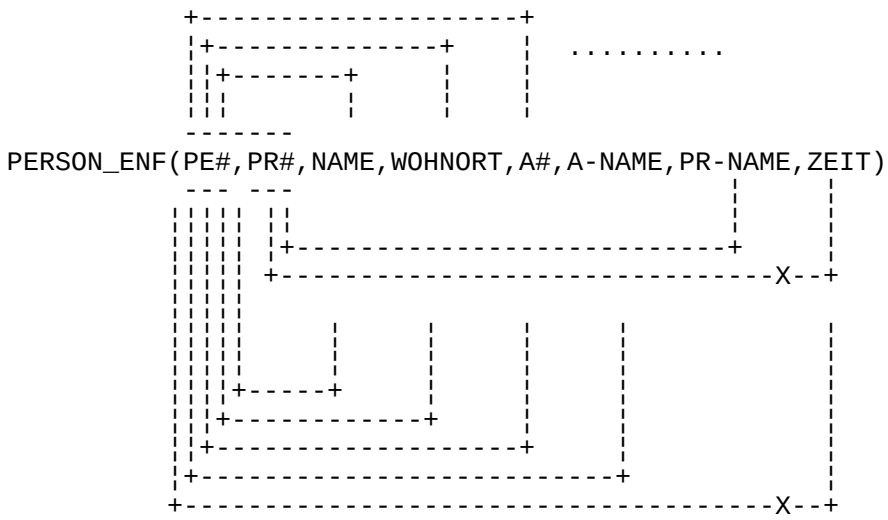


Abb. 2.1-6: Relation PERSON_ENF in erster Normalform

Alle nicht dem Schlüssel (PE#,PR#) angehörenden Attribute sind funktional abhängig vom Gesamtschlüssel (Kriterium der ENF) Nicht dem Schlüssel angehörende Attribute sind funktional abhängig von Schlüsselteilen. Hier ist lediglich ZEIT weder vom Schlüsselteil PE# noch vom Schlüsselteil PR# funktional abhängig (in der vorstehenden Abbildung mit X markiert). Offensichtlich verletzen genau die mit Speicheranomalien in Zusammenhang stehenden Attribute die Kriterien der ZNF.

Aufspalten der Relation PERSON_ENF in ZNF-Relationen

```
PERSON_ZNF(PE#,NAME,WOHNORT,  A#,A-NAME)
          101 Hans Regensburg 1  Physik
          102 Rolf Nürnberg   2  Chemie
          103 Udo  München    2  Chemie
          104 Paul Regensburg 1  Physik
```

PE# repräsentiert hier einen Primärschlüssel

```
PE-PR(PE#,PR#,ZEIT)
      101 11  60
      101 12  40
      102 13  100
      103 11  20
      103 12  50
      103 13  30
      104 11  80
      104 13  30
```

Das Attribut PE# repräsentiert hier einen Fremdschlüssel.

```
PRODUKT(PR#,PR-NAME)
        11   A
        12   B
        13   C
```

Der Fakt kann immer noch verletzt werden und damit zu einer Speicheranamolie führen.

5) DNF

Es ist keine funktionale Abhängigkeit zwischen nicht dem Schlüssel angehörenden Attributen erlaubt.
Funktionale Abhängigkeit besteht in PERSON_ZNF:

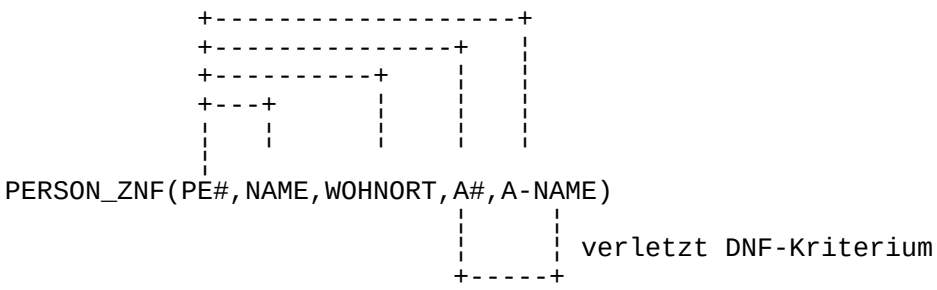


Abb. 2.1-7: Funktionale Abhängigkeiten in PERSON_ZNF

Die Relation PERSON_ZNF wird in DNF-Relationen aufgespalten:

```
PERSON (PE#,NAME,WOHNORT,   A#)
        101 Hans Regensburg 1
        102 Rolf Nürnberg   2
        103 Udo  München    2
        104 Paul Regensburg 1

ABTEILUNG(A#,A-NAME)
           1  Physik
           2  Chemie
```

6) Zusammenfassung

Die Problemstellung wird durch folgende Relationen beschrieben:

```
PRODUKT(PR#,PR-NAME)          PE-PR(PE#,PR#,ZEIT)
        11    A                     101 11  60
        12    B                     101 12  40
        13    C                     102 13  100
                                    103 11  20
                                    103 12  50
```

```
                                    103 13  30
                                    104 11  80
                                    104 13  20

PERSON(PE#,NAME,WOHNORT,A#)
       101 Hans Regensburg 1
       102 Rolf Nürnberg   2
       103 Udo  München    2
       104 Paul Regensburg 1

ABTEILUNG(A#,A-NAME)
          1  Physik
          2  Chemie
```

2.1.3.2 Der Syntheseprozess

Hier wird versucht realitätskonforme Feststellungen mit Hilfe von Elementarrelationen festzuhalten. Diese Elementarrelationen werden dann zur effizienten Verarbeitung systematisch zusammengefaßt. Das Resultat der Kombinationen darf aber kein Normalisierungskriterium verletzen.

1) Problemstellung

Gesucht ist ein Modell, das die Entitätstypen

- PERSON (Mitarbeiter)
- MASCHINE
- PRODUKT

berücksichtigt. Eine Analyse ergab folgende Fakten :

```
1. Fakt                 Ein Mitarbeiter bedient mehrere
                        Maschinen und produziert dabei
2. Fakt                 mehrere Produkte.
                        Eine Maschine wird immer nur von einer
3. Fakt                 Person bedient,
                        kann aber durchaus
4. Fakt                 mehrere Produkte produzieren.
                        Die Herstellung eines Produkts
                        erfordert immer nur
5. Fakt                 eine Maschine sowie
6. Fakt                 eine Person
```

2) Elementarrelationen zur Darstellung der im Problem angeführten Feststellungen

PERSON ist Entitätstyp: PERSON(PE#)
MASCHINE ist Entitätstyp: MASCHINE(M#)
PRODUKT ist Entitätstyp: PRODUKT(PR#)
1. und 3. Feststellung: M-PE(M#,PE#)
2. und 6. Feststellung: PE-PR(PR#,PE#)
4. und 5. Feststellung: M-PR(PR#,M#)

3) Zusammenfassung von Elementarrelationen mit identischen Schlüsseln

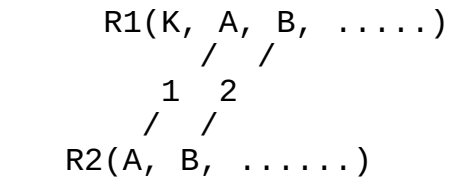
```
PRODUKT(PR#)
M-PR   (PR#,M#)
PE-PR  (PR#,PE#)
--------------------
PRODUKT(PR#,M#,PE#)
            ¦   ¦    Funktionale Abhängigkeit
            +---+     (verletzt DNF)
MASCHINE(M#)
M-PE    (M#,PE#)
--------------------
MASCHINE(M#,PE#)
PERSON(PE#)
```

Normalisierung

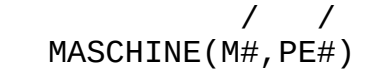
```
PRODUKT(PR#,M#)
M-PE   (M#,PE#)
MASCHINE(M#,PE#)
PERSON(PE#)
```

4) Systematisches Erkennen von Verletzungen der DNF beim Zusammenfassen von Elementaroperationen

Allgemeiner Fall:



Bsp.:



Bedingungen für das Erkennen

1. Die das Kriterium der DNF verletzende Relation R1 (im Bsp. PRODUKT) weist einen Fremdschlüssel A auf (M#) Das bedeutet: Es existiert eine Relation R2 (MASCHINE) in der das Attribut A (M#) den Primärschlüssel darstellt. Ein Attribut einer Relation ist ein Fremdschlüssel in dieser Relation, falls das Attribut nicht Primärschlüssel der Relation ist, aber als Primärschlüssel in einer anderen Relation in Erscheinung tritt.
2. Sowohl R1 (im Bsp. PRODUKT) und R2 (MASCHINE) weisen ein zusätzliches identisches Attribut B (PE#) auf.

Treffen die genannten Bedingungen zu, so ist das Attribut B in der Relation R2 mit Sicherheit vom Attribut A funktional abhängig. Möglicherweise liegt diese funktionale Abhängigkeit auch in der Relation R1 vor. Das würde bedeuten: R1 weist eine transitive Abhängigkeit auf (verletzt DNF)

5) Zusammenfassung

Syntheseschritt 1
Definition eines Modells mit den Konstruktionselementen

- Entitätsmenge
- Entitätsattribute
- Beziehungsmenge
- Beziehungsattribute

und Festhalten dieser Konstruktionselemente mit Hilfe von normalisierten Elementaroperationen.

Syntheseschritt 2
Zusammenfassen von Elementaroperationen mit identischen Schlüsseln

Syntheseschritt 3
In seltenen Fällen fallen bei der Zusammenfassung von Elementaroperationen Relationen an, die die Kriterien der DNF verletzen. Sie können mit Hilfe der genannten Bedingungen systematisch erkannt und einer Normalisierung zugeführt werden.

Hinweis: Nach der Zusammenfassung und der anschließenden Normalisierung können erneut mehrere Relationen mit identischen Schlüsseln anfallen.

Im vorliegenden Bsp. betrifft dies:

```
MASCHINE(M#,PE#)
M-PE     (M#,PE#)
```

Diese Relationen betreffen ein und denselben Sachverhalt (, eine Maschine wird nur von einer Person bedient und ein und diesselbe Person kann mehrere Maschinen bedienen).
Zusammengefaßt ergibt sich: `MASCHINE(M#,PE#)`

2.2 Mathematische Grundlagen für Sprachen in relationalen Datenbanken

2.2.1 Relationenalgebra

2.2.1.1 Die Basis: Mengenalgebra

Da Relationen Mengen sind, muß es möglich sein, Ergebnisse und Begriffe aus der Mengenlehre auf DB-Manipulationen anzuwenden.
Eine DML-Anweisung ist dann: Eine Auswahl einer Untermenge aus einer Menge von Datenelementen.
Die Mengenalgebra stellt dafür Operationen (z.B. Durchschnitt, Vereinigung und Differenz) als Sprachelemente zur Verfügung. Eine Anwendungsmöglichkeit solcher Operationen ist stets auf vereinigungsverträgliche Relationen gegeben.

Definition der Vereinigungsverträglichkeit

- Zwei einfache Attribute sind vereinigungsverträglich, wenn die zu diesen Attributen zugehörigen Werte entweder aus Zahlen oder Zeichenketten bestehen.
- Zwei zusammengesetzte Attribute sind vereinigungsverträglich, wenn sie aus derselben Zahl von Attributen bestehen und die jeweiligen ("j-ten") Attribute vereinigungsverträglich sind.
- Zwei Relationen sind vereinigungsverträglich, wenn zusammengesetzte Attribute, von denen die beiden Relationen Teilmengen bilden, vereinigungsverträglich sind, d.h.: Die Anzahl der Domänen (Wertebereiche, Attribute) von Relationenschema R und S stimmt überein (für alle 1 <= j <= n Wertebereiche stimmt das Format der Daten in dem j-ten Wertebereich von Relation r mit dem j-ten Wertebereich von Relation s überein).

Für vereinigungsverträgliche Relationen sind folgende Mengenoperationen definiert:

1. Die Vereinigung von r und s

führt zu einer Relation, deren Tupel t in r oder s oder in beiden vorkommen:

$\{t \mid t \in r \text{ oder } t \in s \text{ oder beides }\}$

2. Der Durchschnitt von r und s

führt zu einer Relation, deren Tupel t sowohl in r als auch in s vorkommen:

$\{ t \mid t \in r \text{ und } t \in s \}$

3. Die Differenz von r und s

führt zu einer Relation, deren Tupel t in r, nicht aber in s vorkommen.

4. Die symmetrische Differenz von r und s

führt zu einer Relation, deren Tupel t in r oder s, nicht aber in beiden Relationen vorkommen.

Da Datenelemente hier Tupeln einer Menge (normalisierter Relationen) sind, braucht die bekannte Mengenalgebra nur um tupelorientierte Operationen erweitert werden. Voraussetzung dafür ist die Definition einer Tupelvariablen t, der man die Zeile einer Tabelle (Relation) als Wert zuweisen kann. Datenelemente werden aus Tupeln (von Mengen) gebildet. Die bekannten Gesetzmäßigkeiten der Mengenalgebra sind demnach um tupelorientierte Operationen zu erweitern.
Eine solche Algebra ist die **Relationenalgebra**. Auswertungen aus einer Datenbank (DB) sind hier Mengen, die aus Relationen der DB hergeleitet sind. Relationenoperationen können auch Relationen erzeugen, die mit anderen Tupeln als denen der DB ausgestattet sind. Wegen der Universalität und des großen Auswahlvermögens dient die algebraische Sprache häufig sogar als Basissprache (oder zumindest als Vergleichssprache). Relational vollständig heissen Sprachen, die das gleiche Auswahlvermögen wie die Relationenalgebra haben. Damit können alle Fragen in diesen Sprachen ausgedrückt werden, die semantisch in der DB enthalten sind.

2.2.1.2 Operationen der relationalen Algebra

1. Projektion (Entfernen von Spalten aus einer Tabelle)

Die **Projektion** bestimmt aus einer gegebenen Relation r_1 eine zweite Relation r_2 durch Entfernen aller Attribute (Spalten), die nicht speziell als Parameter (bestimmt durch die Namen der Attribute) der Projektion angegeben sind. Nur jeweils eine Ausprägung eines Tupels bei evtl. auftretenden Duplikate wird ausgegeben.
Die Projektion der Relation r [120] kann allgemein dann so beschrieben werden:

$r[A_i,A_j,...,A_m]$
$A_i,A_j,...,A_m$ sind Namen der Attribute.

Bsp.: Operationen zur Projektion

1. Gegeben ist r(A1, A2, A3)

a	bb	5
a	bb	6
a	cc	7

- r[A1]

a

- r[A1,A2])

a	bb
a	cc

- r[A2,A3]

bb	5
bb	6
cc	7

[120] Vereinbarung: $r(A_1, A_2, ... ,A_n)$ ist eine Relation vom Grad n, t ist die zugehörige Tupelvariable; $R = (A_1, A_2, ... , A_n)$ beschreibt das Schema der Relation.

- permutierte Relation

```
r[A2,A1,A3]
  bb a  5
  bb a  6
  cc a  7
```

2. Gegeben ist `r(LNR,TNR)`

```
a   1
a   2
b   1
```

Welche Lieferanten mit welcher Lieferantennummer (LNR) sind in der Lieferantenliste?

Lösung: Projektion von `r` auf `LNR`: `r[LNR]`

2. Selektion (Restriktion)

Hierbei handelt es sich um die Auswahl der Tupel einer Relation, die eine bestimmte Bedingung erfüllen:

$r[A\Theta B]$={t | t ∈ r und (r[A] Θ r[B])}

Voraussetzung: Jedes Element von r[A] ist Θ-vergleichbar[121] mit jedem Element von r[B].

Bsp.: Operationen zur Selektion

```
1. r(A, B, C)
     p  2  1        - r[B = C] (A,B,C)
     q  2  3                    r 3 3
     q  5  4        - r[B > C] (A,B,C)
     r  3  3                    p 2 1
                                q 5 4
```

2. r(TNR,G1,G2)

G1: Teilgewicht mit Verpackung G2: Teilgewicht ohne Verpackung

Welche Teile werden ohne Verpackung geliefert? `(r[G1 = G2]) [TNR]`

3. Verbund

Die Operation Verbund ("join") verknüpft zwei Relationen $r_1(A_1,..,A_n,X)$ und $r_2(X,B_1,...,B_m)$, die ein gemeinsames Attribut besitzen. Mit Hilfe der Verknüpfungs-operationen (=, <>, <, >) wird über die Werte von X eine neue

[121] Θ ist einer der 6 Vergleichsoperatoren

Relation bestimmt, die sowohl alle Attribute aus r_1 und r_2 umfaßt. Falls t_1 in r_1 und t_2 in r_2 enthalten ist, dann umfaßt r_3:

$t_1.A_1, \ldots , t_1.An, t_2.B_1, \ldots , t_2.B_m$

Allgemein schreibt man dann zur Verknüpfung der beiden Relationen über die vereinigungsverträglichen Attribute A und B:

$r_1[A \ \Theta \ B]r_2$

Jedes Element der Spalte r_1[A] ist mit jedem Element der Spalte r_2[B] vergleichbar. Zwei Werte x und y sind Θ-vergleichbar, wenn x Θ y entweder wahr oder falsch ist.
Das Ergebnis eines Verbunds ist dann eine neue Relation, die sich paarweise aus r und s zusammensetzt.
Bsp.:

1. Gegeben ist

```
r(A,B,C) und s(D,E)
  a 1 xx       a 4
  a 2 xx       b 2
  b 1 yy       c 3
  c 3 zz
```

a) Gleichverbund (equi join)

```
r[A = D]s (A, B, C, D, E)
          a  1  xx a  4
          a  2  xx a  4
          b  1  yy b  2
          c  3  zz c  3
```

b) Natürlicher Verbund (natural join)

Das redundante Attribut wird entfernt:

```
(A, B, C, E)
a  1 xx  4
a  2 xx  4
b  1 yy  2
c  3 zz  3
```

c) Verlustfreier Verbund

Keine Information ist in einem Gleichverbund verloren gegangen.

d) Ungleichverbund

```
r[B > E]s (A, B, C, D, E)
          c  3  zz b  2
```

2. Gegeben sind die Relationen r und s

```
r(LNR, NAME)     s(LNR, TNR)
   a   A            a    1
   b   B            a    2
   c   C            b    2
```

c 3

Finde die Namen aller Lieferanten mit den Teile-Nr., die die Lieferanten liefern?

q = r[LNR = LNR]s; Ergebnisrelation: v = q[NAME,TNR]

3. Gegeben ist r(LNR, TNR)

a	1
b	2
b	3
c	4
c	5

Finde die Lieferanten-Nr. von Lieferranten, die die Teile mit derTeile-Nr. 2 und 5 liefern?

Definition von: s(B) = { 2, 5 }
Damit ergibt sich: q = r[TNR = B]s
Eine Projektion q[LNR] führt schließlich auf das Ergebnis.

4. Division

In der elementaren Algebra sind Multiplikation und Division zueinander invers. In der relationalen Algebra ist das **cartesische Produkt** invers zur Division.
Ein wichtige Rolle spielen dabei die Begriffe "Bild" und "Urbild" eines Tupels.

Bsp.: Gegeben sind die Relationen r(R1,R2,R3) und s(S1,S2)

A	B	C	1	2
U	V	W	8	9

Das cartesische Produkt ist: (R1,R2,R3,S1,S2)

A	B	C	1	2
A	B	C	8	9
U	V	W	1	2
U	V	W	8	9

Die Domänen von r sind in der Ergebnisrelation das "Urbild", die Domänen von s das "Bild".

Das **cartesische Produkt** ist eine Verkettung zweier Relationen. Jedes Tupel der einen Relation wird mit jedem Tupel der anderen Relation verknüpft.

Zu einem Tupel der Domäne Urbild kann es mehrere Tupel der Domäne Bild geben. Die Menge dieser Tupel (**Bildmenge**) ist die Bildrelation eines Tupels von Urbild.

Definition

b(x,y) ist eine binäre Relation oder Abbildung. Die **Bildmenge** von x unter b ist dann: $g_b(x) = \{y \mid (x,y) \in b\}$

Bsp.: Bestimmen von Bildmengen gegebener Relationen

1. Gegeben ist: r(A,B)

a	1
b	2
b	3

$g_r(b) = \{2,3\}$; $g_r(1) = \{a\}$

2.Gegeben ist: r(A1,A2,A3,A4)

1	10	x	a
1	10	y	b
2	11	z	b
2	11	x	a
-----	-----		
A		Ä	

Die Projektionen r[A] bzw. r[Ä] ergeben:

r[A](A1 A2)

1	10
2	11

r[Ä](A3,A4)

x	a
y	b
z	b

$g_r(r[Ä]) = g_r(x,a), g_r(y,b), g_r(z,b)$
$g_r(r[Ä]) = \{(1,10),(2,11)\} = r[A]$

Definition der Operation **Division**

(Voraussetzung: r[A] und s[B] sind vereinigungsverträglich)

$r[A : B]s = \{r[Ä] \mid s[B] \subseteq g_r(r[Ä])\}$

(Division von r auf A durch s auf B)

Bsp.: Divisions-Operationen

1. Gegeben ist

r(A, B, U)

1	11	x
2	11	y
3	11	z
4	12	x

s(D, F)

x	1
x	2
y	1

a) r[U : D]s

binäre Relation (U,Ü) mit Ü = (A,B): r[Ü]

1	11
2	11
3	11
4	12

r[U] (U)

x
y
z

Projektion: s[D] = {x,y}

s[D] muß eine Untermenge von g_r(r[Ü]) sein. Das ist in allen 4 Fällen nicht möglich:

$g_r(1,11) = \{x\}$; $g_r(2,11) = \{y\}$; $g_r(3,11) = \{z\}$; $g_r(4,12) = \{x\}$

b) r[B,U][U : D]s

```
r[B,U] (B, U)  r[Ü] (B)   r[U] (U)   s[D] (D)
       11  x         11         x          x
       11  y         12         y          y
       11  z                    z
       12  x
```

$g_r(11) = \{x,y,z\}$; $g_r(12) = \{x\}$;r[B,U][U : D]s = {11}

Aufgaben Gegeben ist

```
r(LNR, TNR) und s(TNR, TB)
   a    1          1    xx
   a    2          2    y
   a    3          3    zz
   b    1
   b    2
   c    3
```

a) Finde die Lieferanten-Nr. der Lieferanten, die alle Teile liefern!

g_r(a) = {1, 2, 3}; s[TNR] = {1, 2, 3}
v = r[TNR : TNR]s = {a}

b) Finde die Lieferanten-Nr. der Lieferanten, die mindestens die Teile mit der Teile-Nr. 1 und 2 liefern!

s(B) = {1,2}; g_r(b) = {1,2}; v = r[TNR : B]s = {a,b}

Hinweis: Die Division in der relationalen Algebra entspricht dem Allquantor im relationalen Kalkül.

5. Nichtalgebraische Operationen

Hierzu gehören vor allen die sogenannten **Aggregatsfunktionen** zum Zählen, Aufsummieren und zur Berechnung des Durchschnitts aus einer Wertemenge.

2.2.1.3 Die Operationen der relationalen Algebra in Standard-SQL (SQL/89)

Nur einige der grundlegenden **Operationen der Relationenalgebra** können über "select-"Anweisungen implementiert werden.

Eine Anweisung der Form

```
SELECT X FROM R;
```

```
(R: tabellen_name, X: Teilmenge der Attribute von R)
```

beschreibt eine **Projektion** von R auf X (doppelt vorkommende Elemente werden nicht beseitigt).

```
SELECT DISTINCT X FROM R;
```

beschreibt eine **Projektion** (im Sinne der Relationenalgebra), d.h.: Mehrfach vorkommende Elemente werden nur einmal aufgeführt.

Eine Anweisung der Form

```
SELECT * FROM R WHERE B;
```

beschreibt eine **Selektion** gemäß der Bedingung B.

Eine Kombination von Projektion und Selektion erhält man dann über

```
SELECT X FROM R WHERE B;
```

Eine Anweisung der Form

```
SELECT * FROM R, S;
```

beschreibt einen **Verbund** von R und S, bei dem keine Verbund-Bedingung angegeben ist. Somit wird hier das kartesische Produkt gebildet. Einen Verbund mit einer Auswahlbedingung (mit Projektion auf X) erhält man durch

```
SELECT X FROM R, S WHERE B;
```

Mengenoperationen können in SQL nicht dargestellt werden. Die Vereinigung kann zwar auf zwei "select-from-where"-Blöcke angewendet werden, z.B.:

```
select ....from ...where ....
union
select.....from ...where ...
```

Sie kann jedoch nicht innerhalb eines "select-from-where-"Blocks auftreten. Differenz und Durchschnitt sind in der aktuellen ANSI/ISO-Fassung von SQL nicht enthalten und müssen innerhalb der "where"-Klausel mit "not" und "and" simuliert werden. Die Operation Division der relationalen Algebra (zur Formulierung des Allquantors) ist ebenfalls in SQL nur über umständliche Simulationen (Negation des Existenzquantors über "not EXISTS") darstellbar.

2.2.2 Das Realtionenkalkül

2.2.2.1 Grundlage: Das Aussagenkalkül

Ein Datenmodell für Datenbanken kann als formale Struktur betrachtet werden, innerhalb der man Aussagen über die reale Welt formulieren kann. Eine Frage an die Datenbank ist dann eine Frage, ob eine bestimmte Aussage zutrifft oder nicht bzw. eine Frage nach denjenigen Objekten, die einer bestimmten Bedingung genügen.
Der **Aussagenkalkül** beschäftigt sich mit Aussagen, denen genau einer der beiden Wahrheitswerte (wahr (w) oder falsch(f)) zugeordnet werden kann. Ein logischer Kalkül besteht aus einer Menge von Axiomem und Ableitungsregeln. Diese Mengen sollen zusammengenommen minimal sein, d.h. keines der Axiome ist aus den übrigen Voraussetzungen ableitbar.
Für die Aussagenlogik erfüllt angeblich[122] das folgende System diese Eigenschaften:

Axiome

(1) $a \rightarrow (b \rightarrow a)$
(2) $(a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$
(3) $(a \rightarrow b) \rightarrow (b \rightarrow a)$

Ableitungsregeln

(1) Für jedes Symbol 'x' kann an jeder Stelle seines Auftretens jeder zu 'x' äquivalente Ausdruck substituiert werden.
(2) Von a und $a \rightarrow b$ kann man zu b übergehen.

Grundlagen der Aussagenlogik sind atomare Formeln (Aussagen). Das sind schlichte, einfache Sätze (Aussagen), die wahr oder falsch sein können., z.B.

josef ist intelligent.

Mit Hilfe der logischen Verknüpfungen (**UND**, dargestellt durch das Symbol $\wedge$; **ODER**, dargestellt durch das Symbol $\vee$; **NOT**, dargestellt durch das Symbol $\neg$; bzw. **WENN ... DANN**, dargestellt durch das Symbol $\rightarrow$) können Verknüpfungen von Aussagen (auf wahr oder falsch) untersucht werden. Dabei benutzt man sog. Wahrheitstabellen, die die Verknüpfung bspw. von 2 Aussagen (abgekürzt durch die Symbole a, b), ob sie wahr oder falsch sind, beschreiben:

a	b	$a \wedge b$	$a \vee b$	$\neg a$
wahr	wahr	wahr	wahr	falsch
wahr	falsch	falsch	wahr	falsch
falsch	wahr	falsch	wahr	wahr
falsch	falsch	falsch	falsch	wahr

[122] vgl.: Schefe, Peter: KI - Überblick und Grundlagen, Mannheim/Wien/Zürich, 1986

a	b	$a \rightarrow b \Leftrightarrow \neg a \vee b$
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	wahr
falsch	falsch	wahr

Abb.: Wahrheitstabellen zu elementaren logischen Operationen

Der Wahrheitswert einer Aussage ist unveränderlich. Aussagenvariable dagegen können verschiedene Wahrheitswerte zugeordnet bekommen. Sie können über logische Verknüpfungen miteinander verbunden werden und bilden dann Aussageformen. Durch Belegung ihre Aussagenvariablen mit Aussagen bzw. Wahrheitswerten (Deutung) erhält man jedoch wieder eine Aussage. Aussageformen können auch durch Verbindungen mit Redewendungen "für alle .... " (Allquantor), "es gibt .... " (Existenzquantor) in Aussagen überführt werden.
Allgemein lassen sich Abfragen an Datenbanken als prädikative Aussageformen formulieren, die durch Einsetzen (von Namen der Objekte) bzw. Verbindungen (mit All-, Existenzquantoren) zu Aussagen (und damit zur Antwort wahr oder falsch) führen.
Die Regeln dazu sind in der Prädikatenlogik der 1. Stufe zusammengefaßt. Viele grundsätzliche Gedanken können aus der Aussagenlogik in die Prädikatenlogik übernommen werden. Die Aussagenlogik beinhaltet wesentliche Grundlagen; sie sind deshalb anschliessend zusammengefaßt

2.2.2.2 Prädikatenlogik

Grundlagen

Die Prädikatenlogik untersucht die innere Struktur von Aussagen. Sie zergliedert einen Satz (Aussagen-) in ein Prädikat und Argumente, z.B.

intelligent(josef)

"`intelligent`" ist das Prädikat, "`josef`" ist das Argument.

vorlesung(juergen, datenbanken)

beschreibt den (Aussagen-) Satz: "Jürgen hält die Vorlesung über Datenbanken". Prädikat ist hier "vorlesung", Argumente sind "`juergen, datenbanken`".

```
ist_intelligent(Student)
```

ist dann eine Aussage, die bestimmt, daß ein Student (, wer auch immer das sein mag ,) intelligent ist. "`Student` ist hier eine **Variable**". Variable beginnen mit einem Großbuchstaben, im Gegensatz zu den Konstanten, die generell mit Zeichenketten aus kleinen Buchstaben benannt werden.
Die Prädikatenlogik untersucht die innere Struktur von Aussagen. Sie teilt den Aussagensatz in Prädikat und Argumente, z.B.:

`Prädikat: Argumente:`

```
abteilung('KO','Konstruktion').[123]
vorlesung(juergen,datenbanken).[124]
abteilung(X,'Konstruktion').[125]
```

ist dann eine Aussage, die dann wahr ist, wenn ein X gefunden (ersetzt) werden kann, zu dem in einer Zeile der Tabelle das Zeichenketten-Literal '`Konstruktion`' passt.

Begriffe aus der Prädikatenlogik

Der Zeichenvorrat der Prädikatenlogik besteht aus:

1. Individuenvariabeln : X,Y,Z,X_1,X_2,....
2. Individuenkonstanten: a,b,c,a_1,b_1
3. Funktionssymbolen: f,g,h,f_1
4. Prädikatensymbolen: p,q,r,r_1
5. Negation ¬
6. Disjunktion und Konjunktion
7. Existenz-Quantor ∃
8. All-Quantor ∀

Die Redewendung "für alle X gilt: ....." (, geschrieben $\forall X:$), heißt **Allquantor**.
Die Redewendung "es gibt ein X, so daß gilt .." (, geschrieben $\exists X:$), heißt **Existenzquantor.**

- ∀ ist eine Abkürzung für mehrere Konjunktionen.
- ∃ ist eine Abkürzung für mehrere Disjunktionen.
- Die Verneinung einer All-Aussage ist eine Existenz-Aussage und umgekehrt.

Im Gegensatz zur Aussagenlogik ist der Wahrheitswert eines prädikatenlogischen Ausdrucks, z.B. der Form $p(X) \vee p(Y)$, nicht ohne weiteres feststellbar, da es sich bei "X" und "Y" um Variablen handelt, die beliebig mit Werten belegt werden können. Erst nach der Belegung der Variablen hat man es wieder mit einem aussagen-logischen Ausdruck zu tun.
Mit Hilfe der Quantoren können allgemeine Aussagen formuliert werden.

"juergen lehrte josel alles".

Die Übersetzung in die Prädikatenlogik lautet:

Für alle X gilt: `lehrte(juergen, josef, X)`, d.h.: Für alle X gilt, daß "juergen es josef" lehrte. Die Variable X umfaßt den Bereich geeigneter Objekte. Sie wird hier durch den Allquantor (symbolische Darstellung ∀) gebunden.

Eine andere Quantifizierung ist

"juergen lehrte josef etwas".

Die Übersetzung in die Prädikatenlogik lautet:

Es gibt ein X: lehrte(juergen, josef, X).

---

[123] mit Zeichenkettenliteralen, die in Anführungszeichen gesetzt werden
[124] Konstante werden generell in der Prädikatenlogik mit kleinem Anfangsbuchstaben angegeben
[125] Variablen definieren einen Großbuchstaben

Die Aussage wird vom Existenzquantor (symbolische Darstellung $\exists$)angeführt und besagt, daß mindestens ein gültiger Wert existiert, den "juergen josef" lehrte.

Terme

1. Jede Individuenvariable ist ein Term.
2. Jede Individuenkonstante ist ein Term
3. $f(t_1,...,t_n)$ ist ein Term, falls f ein n-stelliges Funktionssymbol ist und t_1, ... ,t_n Terme sind.
4. Nur so gebildete Zeichenketten sind Terme

Primformeln

$p(t_1,...,t_n)$ ist eine Primformel, falls p ein n-stelliges Prädikatsymbol ist und t_1 ... t_n Terme sind.

Abkürzungen und Bindungsregeln

- $\alpha \rightarrow \beta$ bezeichnet $\neg\alpha \vee \beta$
- $\alpha \leftrightarrow \beta$ bezeichnet $(\alpha \rightarrow \beta) \wedge (\alpha \leftarrow \beta)$
- $\forall, \exists$ binden stärker als $\neg$
- $\neg$ bindet stärker als $\wedge$
- $\wedge \cdot$ bindet stärker als $\vee$

Formeln

1. Primformeln sind Formeln
2. $\neg\alpha$ ist eine Formel, falls α eine Formel ist
3. $(\alpha \vee \beta)$ ist eine Formel, falls α und β Formeln sind
4. $(\alpha \vee \beta)$ ist eine Formel, falls α und β Formeln sind
5. $\exists X \alpha$ ist eine Formel, falls α eine Formel und X eine Individuenvariable ist
6. $\forall X \alpha$ ist eine Formel, falls α eine Formel und X eine Individuenvariable ist.
7. Nur so gebildete Zeichenketten sind Formeln.

Sind in einer Formel α alle Variablen durch Quantoren gebunden, wird α auch als geschlossene Formel bezeichnet.

Bsp.: Die Formel $\forall X(p(X) \rightarrow \exists Y(q(Y) \wedge r(Y,X)))$ ist zu lesen: "Für alle X gilt: Wenn p von X, dann gibt es mindestens ein Y für das gilt: q von Y und r von X und Y."
Wird für die Prädikatensymbole festgelegt „p = "`ist_geboren`", q = "`ist_weiblich`", r = "`ist_die_Mutter_von`" “, dann erhält die Formel die Bedeutung: Für alle X gilt: Wenn X geboren wurde, dann gibt es mindestens ein Y, das weiblich ist und dieses Y ist die Mutter.

Zentrale Bedeutung haben hier die sog. **Horn**-Formeln der Form $Q\alpha \rightarrow \beta$. "Q" ist eine Folge von Allquantoren. α bezeichnet man als Prämisse (Voraussetzung), β ist dann die Konklusion (Schlußfolgerung).

Interpretation prädikatenlogischer Formeln

Eine Interpretation ordnet Individuenkonstanten Individuen der realen Welt zu, Prädikaten Mengen von Individuen (z.B. dem Prädikat "menschlich" die Menge aller Menschen). Eine Interpretation beinhaltet

1. einen (nichtleeren) Individuenbereich (Wertebereich der Individuenvariablen)
2. eine Zuordnung
   - eines Elements aus dem Individuenbereich zu einem Konstantensymbol (z.B.: a bedeutet die Zahl 3)
   - eines n-stelligen Funktionssymbols zu einer auf dem Individuenbereich definier-ten Funktion mit n Argumenten (z.B.: f bedeutet die Addition von 2 Zahlen, f(x,y) ist dann x + y)
   - jedes n-stelligen Prädikatensymbols zu einem im Individuenbereich definierten n-stelligen Prädikat, das jedem n-Tupel von Individuen einen Wert aus {wahr, falsch} zuordnet (z.B.: p(x,y) bedeutet x < y)

Bsp.: Gegeben ist der Individuenbereich {0, 1, 2}, f(x,y) mit der Bedeutung "Das kleinere der beiden Argumente x und y (bei Gleichheit x)", das Prädikat q(x,y) mit der Interpretation "x ist gleich y", die Konstante a. Ihr ist das Objekt 2 zugeordnet.
Wie wird dem Ausdruck $\forall X:(\neg q(X,a) \rightarrow q(f(X,a),X))$ ein Wahrheitswert zugeordnet?

Einsetzen für Konstantensymbol a: $\forall X:(\neg q(X,2) \rightarrow q(f(X,2),X))$
Einsetzen für die gebundene Variable X:
$(\neg q(0,2) \rightarrow q(0,0)) \wedge (\neg q(1,2) \rightarrow q(1,1)) \wedge (\neg q(2,2) \rightarrow q(2,2))$
Anwendung des Prädikats: $(\neg falsch \rightarrow wahr) \wedge (\neg falsch \rightarrow wahr) \wedge (\neg falsch \rightarrow wahr)$
Auswertung: $(wahr \rightarrow wahr) \wedge (wahr \rightarrow wahr) \wedge (wahr \rightarrow wahr) = wahr \wedge wahr \wedge wahr$

Offenen Formeln mit freien Variablen kann ebenfalls ein Wahrheitswert nach der vorliegenden Interpretationsvorschrift zugeordnet werden. Vor Interpretationsschritt 2 (Zuordnung) ist noch zu beachten: Jedes Auftreten der freien Variablen ist durch ein Individuum aus dem Individuenbereich zu ersetzen, z.B. ergibt der Ausdruck $\forall X:(\neg q(X,Y) \rightarrow q(f(X,Y),X))$ mit den vorstehenden Interpretationen

- für Y = 0 den Wert falsch
- für Y = 1 den Wert falsch
- für Y = 2 den Wert wahr

Es kann sehr unterschiedliche Interpretationen von Formeln geben, z.B.: Die Formel $\forall X: p(f(X,a),X)$ besitzt 2 Interpretationen:

1. Interpretation

- Der Individuenbereich ist die Menge der natürlichen Zahlen
  Der Konstanten a wird die Zahl 1 zugeordnet
- Der zweistellige Funktor f steht für die Multiplikation
- Das zweistellige Prädikat steht für "=" (Gleichheit natürlicher Zahlen)

Welche Aussage ist dann durch die angegebene Formel bestimmt?

Für alle natürliche Zahlen, die anstelle von X eingesetzt werden, gilt $X \cdot 1 = X$

2. Interpretation

- Der Individuenbereich ist die Menge der ganzen Zahlen
- Der Konstanten a wird der Wert 2 zugeordnet
- Der zweistellige Funktor steht für "+" (Addition auf ganzen Zahlen)
- Das zweistellige Prädikat p steht für die Relation ">"

Welche Aussage ist dann durch die angegebene Formel festgelegt?
Für alle ganze Zahlen, die anstelle von X eingesetzt werden, gilt: $X + 2 > X$

Eine Interpretation, die einer geschlossene Formel eine wahre Aussage zuordnet, heißt **Modell** für diese Formel. Eine Formel heißt **erfüllbar**, wenn es ein Modell für diese Formel gibt (,andernfalls heißt sie unerfüllbar). Ein Formel heißt **allgemeingültig**, wenn jede Interpretation dieser Formel ein Modell ist.
Allgemein betrachtet, ist folgende Vorgehensweise zur Bestimmung des Wahrheitsgehalts eines logischen Satzes angebracht: Ausgangspunkt ist ein Konzept eines Ausschnitts der realen Welt und eine Menge von Sätzen der Prädikatenlogik. Symbole der logischen Sätze werden mit Objekten, Beziehungen, Funktionen (der Konzeptualisierung) in Verbindung gebracht, d.h. logische Symbole bekommen eine Bedeutung. Danach werden den logischen Sätzen durch eine Feststellung Wahrheitswerte zugewiesen. Sie sind genau dann wahr, wenn sie das Konzept der realen Welt korrekt beschreiben. Sonst sind die Sätze falsch.

Klauseln sind Ausdrücke der Form

$$b_1,...,b_m \leftarrow a_1,...,a_n$$

wobei $b_1, ... , b_m, a_1, ... , a_n$ atomare Formeln sind (m >= 0, n >= 0). Die a_i sind die konjunktiv verknüpften Bedingungen der Klauseln, die b_i sind die **Konklusionen**. Enthält die Klausel Variable $X_1, ... , X_k$, dann läßt sich die Klausel so interpretieren:

Für alle $X_1, .. ,X_k$ gilt: $b_1 \vee ... \vee b_m \leftarrow a_1 \wedge ... \wedge a_n$.
Falls n = 0 ist (, keine Bedingung ist spezifiziert), dann ist die Interpretation: Für alle $X_1, .. , X_k$ gilt: $b_1 \vee ... \vee b_m$
Falls m = 0 ergibt sich: Für alle $X_1, .. ,X_k$ gilt: $a_1 \wedge ... \wedge a_n$ ist falsch.
Für m = n = 0 erhält man die leere Klausel, die immer dem Wert falsch entspricht.

Horn-Klauseln sind Klauseln (bzw. Clausen), auf deren linke Seite nur eine einzige atomare Formel stehen darf.

$$b_1 \leftarrow a_1,...,a_n$$

Resolutionsprinzip

Logik kann die Grundlage für automatische Theorembeweis für logisches Programmieren und für die Darstellung von Wissen sein. Dies sind verschiedene, aber verwandte Aktivitäten. Sie beinhalten den Mechanismus der **Inferenz**, d.h.: Wichtig ist vor allem die Frage, welche Schlüsse man aus einer gegebenen Menge von Klauseln ziehen kann und welche Schlüsse zu einer Lösung eines gegebenen Problems führen.
Theorembeweiser benutzen nur eine einzige Inferenzregel: die **Resolution** (1965 von Robinson beschrieben). Sie kann als Verallgemeinerung von bekannten Regeln betrachtet werden, z.B.

- dem **Modus Ponens**
  (, der bspw. b aus a $\rightarrow$ b und a ableitet)
- dem **Modus Tollens**
  (, der bspw. $\neg$a aus a $\rightarrow$ b und $\neg$b ableitet)
- der **Kettenrege**l
  (, die bspw. a $\rightarrow$ c aus a $\rightarrow$ b und b $\rightarrow$ c ableitet)

Beim Resolutionsverfahren wird beweismethodisch ein Widerspruchsverfahren geführt. Gelingt aus der Negation der zu beweisenden Klauseln ein Widerspruch herzuleiten, so ist der Beweis gelungen. Der Widerspruch ist festgelegt, sobald die leere Klausel abgeleitet ist.

2.2.2.3 Logische Systeme: Prolog

Grundlagen

Grundlage von Prolog (vor allem der in Prolog implementierten Logiksprache) ist eine Teilmenge der Prädikatenlogik, die **Horn-Logik**. Prolog-Programme sind Kon-junktionen von sog. Programm-Klauseln (universelle, definierte Horn-Formeln, definite clauses). **Programm-Klauseln** sind Implikationen von nichtnegierten Konditionen (Bedingungen) und einer nichtnegierten Konklusion. Eine **Horn-Klausel** ist eine spezielle Klausel:

$$b_1 \leftarrow a_1,...,a_n$$

b_1 ist die Konklusion der Horn-KLausel. Enhält die Horn-KLausel Variablen X_1, ... , X_k, so ist das so zu interpretieren:

Für alle X_1 ... X_k gilt: $b_1 \leftarrow a_1 \wedge ... \wedge a_n$

Vier Fälle sind zu unterscheiden :

(1) m = 1, n = 0

$b_1 \leftarrow$

Die ist eine einfache atomare Formel, die von keiner Bedingung abhängig ist. In Prolog schreibt man bspw.

```
teil(e1,einzelteil_1).
struktur(p1,b1,2).
```

Jede Klausel wird in Prolog durch einen Punkt abgeschlossen. Klauseln, die durch Fall (1) beschrieben werden, sind **Fakten**.

(2) m = 1, n <> 0

$b_1 \leftarrow a_1 \wedge ... \wedge a_n$

Dies ist der übliche DANN - WENN - Fall.

In Prolog schreibt man

- anstelle von "$\leftarrow$" das Symbol "**:-**"
- anstelle von "$\wedge$" ein Komma.

In dieser Form beschreibt Prolog **Regeln**. Das folgende Prädikat definiert eine derartige Regel:

```
grossvater(Y,X) :- vater(Y,Z), vater(Z,X).
```

Das bedeutet: X ist der Großvater von Y, wenn Z der Vater von Y und X Vater von Z ist. Das Prädikat `grossvater(Y,X)` ist nur beweisbar, wenn die beiden Fakten `vater(Y,Z)` und `vater(Z,X)` vorliegen.

Regeln haben in Prolog folgendes allgemeines Format:

```
schlussfolgerung :- bedingung_1, bedingung_2, ....
```

Eine **Regel** besteht formal aus dem Regelkopf und dem Regelrumpf. Beide sind verbunden über das Symbol „**:-**“.
Die Schlußfolgerung ist dann wahr, wenn alle Bedingungen des Regelrumpfes erfüllt sind.
Regeln definieren den Zusammenhang, in dem die Fakten eines Prolog-Programms interpretiert werden. Regeln und Fakten bilden die Wissensbasis des Programms.

(3) m = 0, n <> 0

$\leftarrow a_1 \wedge ... \wedge a_n$

Die Formel besteht aus Bedingungen. Sie kann so interpretiert werden:

Es ist nicht der Fall, daß gilt $a_1 \wedge ... \wedge a_n$

Diese Ausprägung einer Klausel gibt es in Prolog in der Form einer **Anfrage**. Allerdings wird hier das Symbol "$\leftarrow$" durch ein "**?-**" ersetzt.
Anfragen leiten aus Fakten und Regeln Antworten ab. Prolog interpretiert die Horn-klauseln prozedural: Eine Anfrage löst einen Aufruf eines Fakts bzw. des Regelkopfs aus.

(4) m = 0, n = 0

$\leftarrow$

Diese **leere Klausel** erscheint im Prolog-Programm selbst nicht. Sie ist beweis-technisch[126] wichtig. Der Ablauf des Programms besteht in dem Versuch, ein Faktum aus dem Programm (der Wissensbasis aus Fakten und Regeln) abzuleiten. Die Ableitung erfolgt nach einem fest vorgegebenen, in Prolog implementierten Schlußfolgerungsmechanismus. Herzuleitende Fakten werden vom Benutzer in der Form von Implikationen ohne Konklusionsteil (Anfrage, goal) an den Inferenz-mechanismus übergehen, z.B.

?-struktur(p1,b2,X).

mit der Bedeutung: "Gibt es ein X, so daß struktur(p1,b2,X) aus dem Programm folgt". Der in Prolog eingebettete Schlußfolgerungsmechanismus (Resolution) ver-

[126] vgl. Resolutionsverfahren, 2.5.1

sucht herzuleiten, daß die Anfrage mit den Formeln des Programms im Widerspruch steht, wenn für X bestimmte Werte eingesetzt werden.

Bsp.: Gegeben ist das folgende Prolog-Programm

```
kind_von(juergen,christian).              /* 1. Fakt */
kind_von(juergen,liesel).                 /* 2. Fakt */
mann(christian).                          /* 3. Fakt */
mann(juergen).                            /* 4. Fakt */
frau(liesel).                             /* 5. Fakt */
mutter_von(X,Y) :-                        /* 1. Regel */
 kind_von(Y,X), frau(X).
vater_von(X,Y) :-                         /* 2. Regel */
 kind_von(Y,X), mann(X).
```

An dieses Prolog-Programm wird die Anfrage

```
?-mutter_von(liesel,juergen).
```

gestellt.

Die Frage ist durch den Kopf der ersten Regel ersetzbar. Prolog durchsucht die Wissensbasis vom Anfang bis zum Ende nach einem passendem Fakt bzw. einem passenden Regelkopf. Die 1. Regel ist an die vorliegende Regel angepaßt, wenn "X durch liesel (X/liesel)", "Y durch juergen (Y/juergen)" ersetzt wird. Diesen Vorgang nennt man in Prolog **Unifikation**. **Unifikation** heißt: Prüfen, ob Literale zusammenpassen!

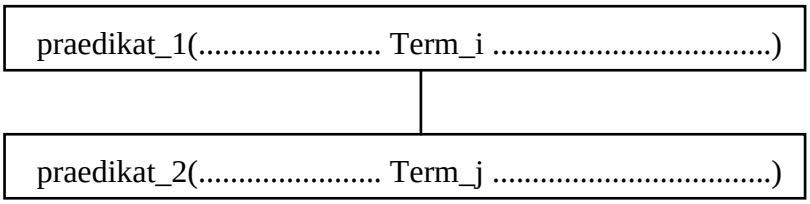


Abb. : Unifikation zweier Prädikate

Etwas vereinfacht besteht **Unifikation** aus 3 Prüfungen:

1. Passen die Prädikate (praedikat_1 = praedikat_2)
2. Stimmen die Anzahl der Terme überein
3. Passen Terme paarweise.

Wenn eine Anfrage beantwortet werden soll, wird in der Wissensbasis (Prolog-Pro-gramm) nach einem Faktum bzw. dem Kopf einer Regel gesucht, das bzw. der mit der Anfrage unifiziert (verschmilzt). Anfrage und Klauselkopf unifizieren, wenn die der Unifikation zugrundeliegenden Prüfungen ergeben: Es ist möglich, Variablen so zu ersetzen, daß die beiden Ausdrücke gleich werden.

Die 1. Regel im vorliegenden Beispiel verweist im Regelrumpf auf weitere Teilziele, die erfüllt sein müssen. Es gilt zunächst das Teilziel "kind_von(juergen,liesel)" zu beweisen. Prolog durchsucht zu diesem Zweck wieder die Wissenbasis vom Anfang bis zum Ende und stößt dabei auf den 2. Fakt. Ein Fakt ist immer wahr, das Teilziel ist erfüllt. Es folgt die Realisierung des weiteren Teilziels "frau(liesel)". Prolog durchsucht die Wissensbasis vom Anfang bis zum Ende und findet eine Bestätigung des Teilziels im 5. Fakt. Die Anfrage wurde damit vollständig bewahrheitet. Der Prolog-Interpreter bestätigt dies durch die Antwort: YES.

Den geschilderten Ablauf kann mit Hilfe einer graphischen Darstellung (Beweisbaum) zusammenfassen. Zur Verdeutlichung des prädikatenlogischen Resolutionsbeweises, werden die Regeln in eine äquivalente Darstellung mit disjunktiv, verknüpften atomaren Formeln überführt. Es ergibt sich

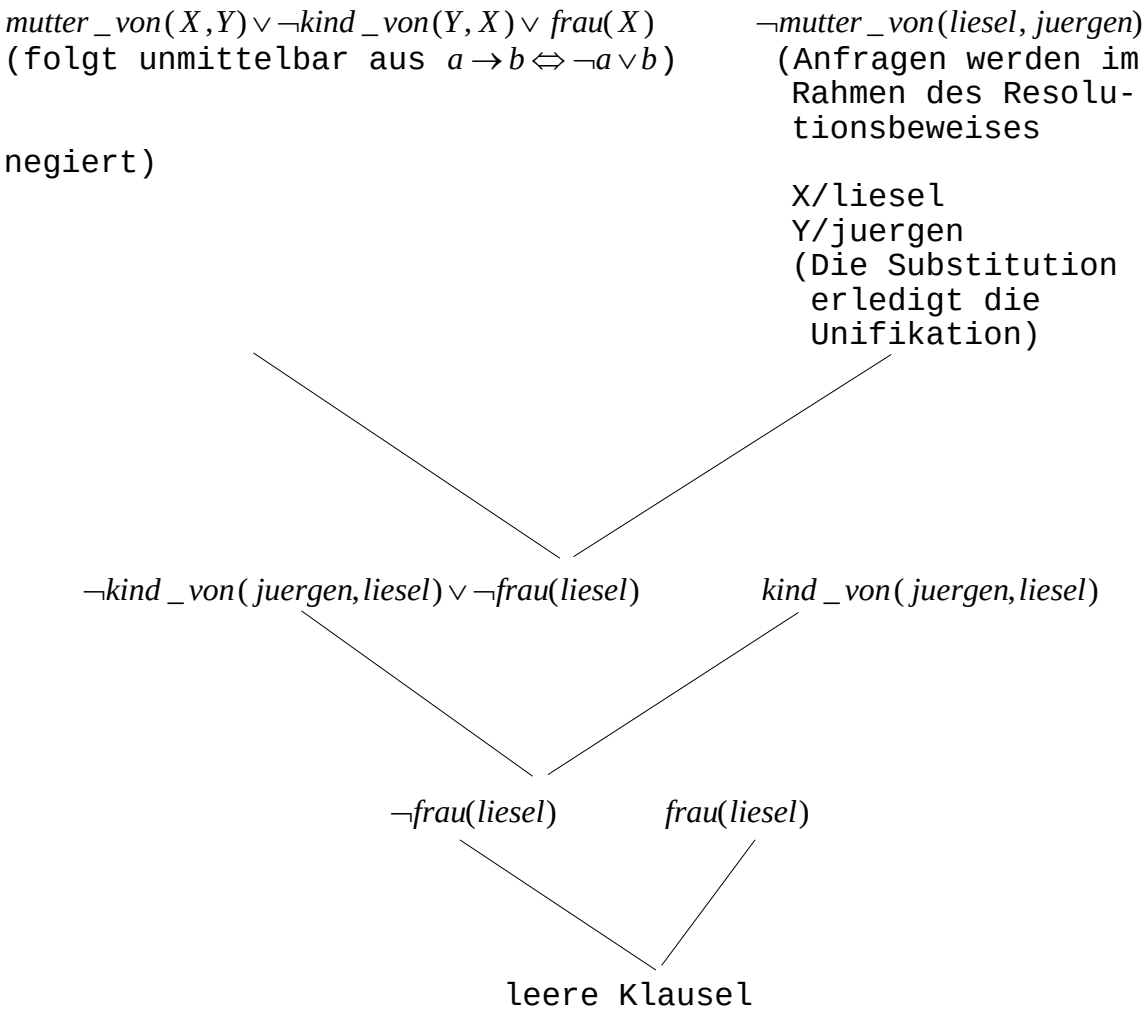


Abb. : Beweisbaum zur Zielvorgabe "?-mutter_von(liesel,juergen)."

Die leere Klausel ist immer falsch, die Anfrage somit wahr.

Der pädikatenlogische Resolutionsbeweis zu der folgenden Anfrage

```
?-vater_von(X,Y).
```

kann ebenfalls direkt im Beweisbaum gezeigt werden:

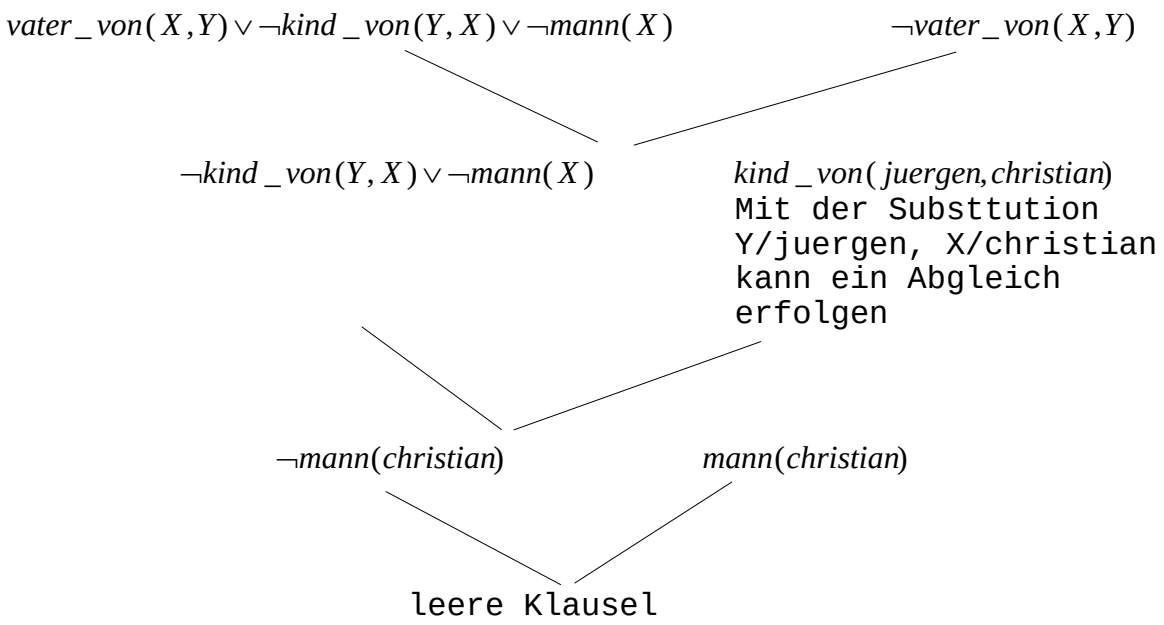


Abb. : Beweisbaum zur Zielvorgabe "?-vater_von(X,Y)."

Der Beweis konnte nur durch die Zuordnung Y/juergen bzw. X/christian gelingen. Man spricht von **Instanziierung der Variablen.**Prolog gibt an, durch welche Bindung der Variablen der Beweis gelungen ist:

```
X=christian
Y=juergen
```

Prolog durchsucht Fakten und Regeln immer von links nach rechts. Da links das Ergebnis und rechts die Konjunktion von Ursachen steht, liegt eine **rückwärtsver-kettende Kontrollstruktur** vor.

Aufgabe

Zu welchem Ergebnis führt die Anfrage

```
?-p(X).
```

an das folgende Prolog-Programm:

```
p(X) :- q(X).      /* 1. Regel */
q(X) :- p(X).      /* 2. Regel */
q(a).              /* Fakt */
```

Die Anfrage führt zu keinem Ergebnis, sondern zu einer Endlosschleife.

Begründung: Um p(X) zu beweisen, ist q(X) zu beweisen (/* 1. Regel */). q(X) könnte mit X=a bewiesen werden, zuerst wird hier immer die Regel q(X) :- p(X) aufgerufen, dh.: Um q(X) zu beweisen ist wieder p(X) zu beweisen. Das Programm erreicht also wegen dieser Anordnung und dem vorgegebenen, in Prolog fest implementierten Ablauf (von oben nach unten, von links nach rechts) nie den Kandidaten q(a) und gerät in eine Endlosschleife.
Nach welcher Änderung im vorstehenden Programm kann ein korrekter Ablauf erzwungen werden?

```
p(X) :- q(X).
q(a).
q(X) :- p(X).
```

Fakten, Regeln, Anfragen bestehen aus Prädikaten, das sind logische Funktionen, die immer nur die Werte "wahr" und "falsch" annehmen können. In Prolog werden Prädikate so beschrieben:

```
praedikat(... Term ....).
```

Ein **Term** kann eine Konstante, eine Variable eine beliebig geschachtelte Struktur oder eine Liste sein.
Die **syntaktischen Elementarformen** von Prolog sind **Terme**. Jede Verknüpfung von Termen ergibt wiederum Terme. Diese rekursive Definition ermöglicht, daß Prolog den Term als einzige syntaktische Form kennt. Zusammengesetzte Terme heißen **Strukturen**. **Variable** beginnen mit einem Großbuchstaben. Ausgenommen davon ist die **anonyme Variable**. Sie ist durch das Symbol "_" gekennzeichnet und definiert eine Variable, deren Wert nicht von Belang ist. **Konstanten** sind Zahlen oder Zeichenketten, die mit Kleinbuchstaben beginnen oder in Anführungszeichen eingeschlossen sind.
Der **lexikalische Geltungsbereich** von Variablennamen ist eine Klausel. Der gleiche Name, der in 2 Klauseln auftritt, bezeichnet 2 verschiedene Variable. Jedes Auftreten einer Variable in derselben Klausel bezeichnet die gleiche Variable. Konstante dagegen verhalten sich anders: Die gleiche Konstante bezeichnet immer das gleiche Objekt in jeder beliebigen Klausel, d.h. im ganzen Programm.
Bsp.:

```
7, 4, fachhochschule,          sind Konstanten
Was, Wie, Kopf, Rest           sind Variable
```

Standardisierung

Für **Prolog** existiert, das haben die vorstehenden Beispiele gezeigt, kein Standard. Die am häufigsten verwendete Syntax wurde an der Universität Edinburgh entwickelt und in dem bekannten Werk von Clocksin / Mellish[127] (C&M-Standard-Prolog) beschrieben. Sie wird manchmal als "de facto" - Standard bezeichnet. Es existieren aber zahlreiche Versionen von Prolog, die zum Teil erheblich von dieser Syntax abweichen.
Prolog ist keine einheitliche Programmiersprache, sondern eine Programmier-konzeption. Die syntaktischen Unterschiede zwischen existierenden Prolog-Systemen sind weitaus größer als z.B. die zwischen C und Pascal.

Ein Prolog-Programm, das eine relationale Datenbank bearbeitet

Die folgende Fakten-Zusammenstellung beschreibt eine relationale Datenbank:

```
teil(e1,einzelteil_1).
teil(e2,einzelteil_2).
teil(e3,einzelteil_3).
teil(b1,baugruppe_1).
teil(b2,baugruppe_2).
```

[127] Clocksin, W.F. und Mellish, C. S.: "Programming in Prolog, Second Edition, Berlin/Heidelberg/New York/Tokio, 1984

```
teil(p1,endprodukt_1).
teil(p2,endprodukt_2).
struktur(p1,b1,2).
struktur(p1,b2,3).
struktur(p1,e3,10).
struktur(p2,b1,3).
struktur(p2,e3,8).
struktur(b1,e1,7).
struktur(b1,e2,8).
struktur(b2,e2,10).
struktur(b2,e3,4).
```

Ein Tupel in einer relationalen Datenbank entspricht einem Prolog-Fakt. Der Name der Tabelle (die Relation) kann als Name des Prädikats gewählt werden, die Spalten der Tabelle entsprechen den Positionen der Argumente des Prädikats.
Im relationalen Datenmodell sind keine Regeln vorgesehen. Das bedeutet nicht: Regeln sind in der Datenbankwelt generell überflüssig. Falls ein Datenbanksystem Fakten und Regeln verarbeiten kann, dann können Daten u.U. sehr vorteilhaft organisiert werden. Derartige Vorteile sind:

- das Einbringen und Ändern von Daten wird erleichtert (z.B. durch Errechnen von Tabellenwerten)
- es wird weniger Speicherplatz verbraucht
- Sichten (views) für spezifische Benutzer sind leichter zu definieren
- Integritätsbedingungen sind einfach zu definieren.

Die 3 grundlegenden Basisabfragen einer relationalen Datenbank (vgl. 2.2.3) lassen sich mit dem vorliegenden Prolog-Programm einfach realisieren. So ist

```
?-struktur(X,Y,Z).
```

eine Anfrage, die alle Zeilen der Tabelle liefert:

```
X=p1, Y=b1, Z=2
X=p1, Y=b2, Z=3
X=p1, Y=e3, Z=10
etc.
```

```
?-struktur(X,Y,10).
```

ist eine Anfrage, die nur einige Zeilen der Tabelle bereitstellt:

```
X=p1, Y=e3
X=b2, Y=e2
```

```
?-teil(_,Bezeichnungen).
```

ist eine Anfrage, die nur die Spalten der Tabelle teil liefert, z.B.:

```
Bezeichnungen=einzelteil_1
Bezeichnungen=einzelteil_2
etc.
```

```
?-struktur(X,Y,Z),teil(X,Name_1),teil(Y,Name_2).
```

ist eine Anfrage, die Ausgaben der folgenden Form zeigt:

```
X=b2, Y=e3, Z=4, Name_1=baugruppe_2, Name_2=einzelteil_3
```

Die Anfrage realisiert demnach den Verbund 2er Tabellen.

Herzuleitende Fakten werden vom Benutzer in der Form von Implikationen ohne Konklusionsteil (Anfrage, goal) an den Inferenzmechanismus übergeben, z.B.:

```
?struktur(p1,b2,X)
```

mit der Bedeutung: "Gibt es ein X, so daß "struktur(p1,b2,X)" aus dem Programm folgt. Der in Prolog eingebettete Schlußfolgerungsmechanismus versucht herzuleiten, daß die Anfrage mit den Formeln des Programms in Widerspruch steht, wenn für X bestimmte Werte eingesetzt werden. Der Reihe nach geschieht:
Für jedes Ziel wird von oben nach unten das Prolog-Programm (Wissensbasis aus Fakten und Regeln) der Abgleich (matching) versucht:

- Innerhalb der Regeln werden Teilziele von links nach rechts abgearbeitet. Eine **Tiefensuche** wird eingeleitet. Ist ein Teilziel wieder ein Regelkopf, wird erst diese Regel, die evtl. weitere Regelköpfe als Teilziele enthalten kann, abgearbeitet.
- Führt ein Teilziel nicht zum Erfolg, werden die Instanziierungen, die bei diesem Teilziel erfolgten, gelöst. Es folgt ein Zurückgehen zum vorliegenden Teilziel (**Backtracking**).
- Ist ein Faktum mit passendem Prädikat und passender Stelligkeit gefunden, so tritt der Prozeß der Beantwortung einer Frage in die Phase der Unifikation ein. Es wird versucht, ein Ziel mit einer Klausel zur Deckung zu bringen ("matching", ent-spricht grob gesehen der Parameterübergabe in Programmen prozeduraler Programmiersprachen)

Backtracking

Das wiederholte Zurückgehen in die Datenbasis ist die wichtigste Kontrollstrategie von Prolog. Sie heißt **Backtracking**. Dabei werdem bestehende Variablen-Instanziierungen aufgelöst, und die Suche beim letzten markierten Punkt (Choicepoint) der Wissensbasis fortgeführt.
Das **Backtracking** innerhalb eines Prolog-Programms läßt sich mit dem sog. Vierport-Modell eines Prädikats veranschaulichen:

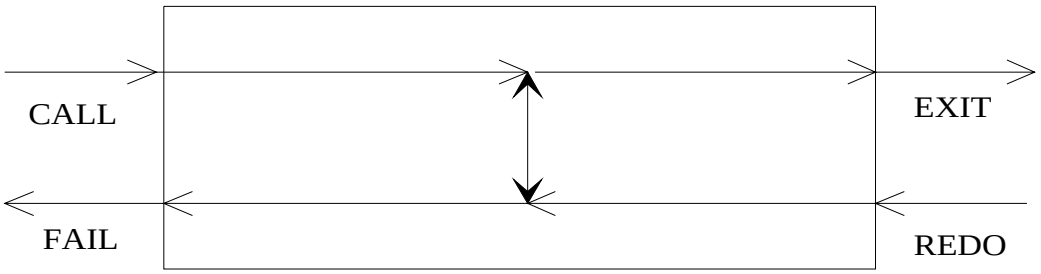


Abb. : Vierport-Modell eines Prädikats

Das Backtracking steuert den Informationsfluß zwischen den Aus- und Eingängen der Klauseln:

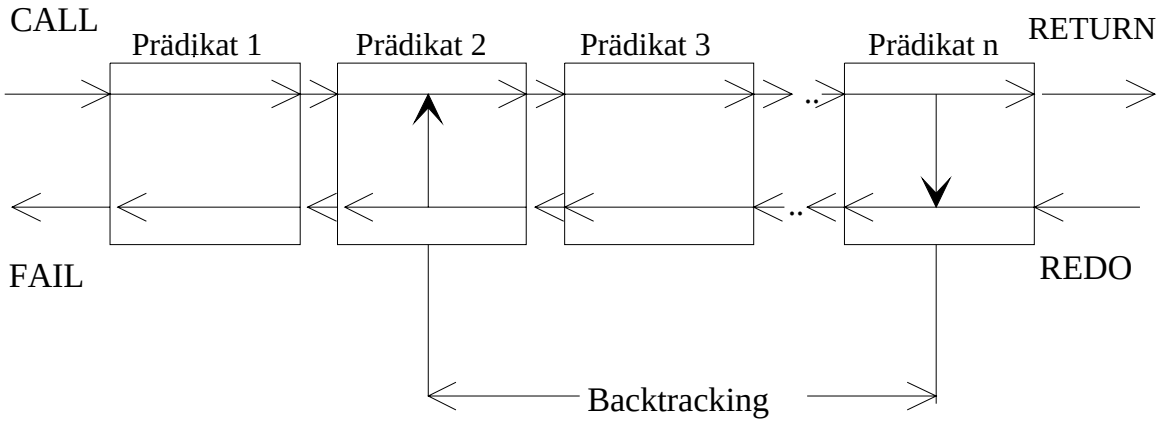


Abb. Informationsfluß zwischen Ein-/Ausgängen der Klauseln

Das Backtracking beruht darauf, daß bei Unifikation eines Ziels mit dem Klauselkopf häufig noch Alternativen bestehen, d.h.: Es können auch noch andere Klauselköpfe mit dem Ziel unifizieren. Solche Zustände, zusammen mit den bis dahin durchgeführten Variablenbindungen, werden Choicepoints genannt. Bei einer Fehlanzeige kehrt der Interpreter zum zuletzt besuchten Choicepoint zurück und versucht den Beweis des Ziels mit einer alternativen Klausel. Die Rückkehr ist verbunden mit der Freigabe der Variablen, die seit dem ersten Anlaufen des Choicepoint instanziiert wurden.

Bsp.: Gegeben sind Fakten eines Prolog-Programms, das eine Tabelle einer relationalen Datenbank beschreibt:

```
boss_von(josef,anton).
boss_von(juergen,josef).
boss_von(anna,maria).
boss_von(maria,anton).
```

Eine Anfrage soll den Superboss ermitteln:

```
?-boss_von(X,Y),
boss_von(Y,Z).
```

1. Der 1. Fakt der Wissensbasis wird mit der 1. Teilabfrage abgeglichen. Das führt zu `X=josef, Y=anton`
2. Der 2. Teil der Anfrage startet eine Suche nach einem Fakt mit dem 1 Argument "anton". Da es einen solchen Fakt nicht gibt, kommt es zur Fehlanzeige.
3. Backtracking führt zu einer anderen Auswahl. Der 2. Fakt der Wissensbasis wird mit der 1. Teilabfrage abgeglichen und führt zum Ergebnis:
   `X=juergen, Y=josef`
4. Die Suche nach einem Fakt mit dem 1. Argument "josef" ist erfolgreich.
5. Es ergibt sich Z=anton und damit ist die Abfrage erfolgreich abgewickelt.
6. Es wird nach einem weiteren Fakt gesucht, dessen 1. Argument "josef" ist
7. Einen solchen Fakt gibt es nicht.
8. Damit ist auf die 1. Teilabfrage zurückverzweigt worden. Es kann ein neuer Versuch gestartet werden. Der erneute Abgleich führt auf
   `X=anna, Y=maria`
9. Es wird jetzt nach einem Fakt gesucht, dessen 1. Argument "maria" ist. Ein solcher Fakt existiert und das führt zum Ergebnis
   `Z=anton`

Mit Hilfe des **Backtracking** können nacheinander alle Objekte erzeugt werden, die irgendein Ziel erfüllen.

Backtracking muß manchmal u.U. auch verhindert werden. Dies geschieht mit Hilfe des "**cut**" (, abgekürzt durch das Zeichen **!**).

Zwei Fälle sind zu unterscheiden:

1. Steht der cut am Ende der Regel und das System erreicht den cut, dann ist das Prädikat erfolgreich und beim Backtracking werden keine alternativen Lösungswege für das Backtracking gesucht.
2. Stehen hinter dem "cut" noch weitere Ziele, so führt Backtracking innerhalb der Regel nur zum "cut" aber nicht weiter zurück. Außerdem ist dann die Entscheidung, welche Klausel für das Prädikat geprüft werden soll, eingefroren. Damit ist der Aufruf des Prädikats, in dem der cut programmiert ist, falsch.

Der Cut sagt Prolog, wie es ein bestimmtes Ziel beweisen soll. Es handelt sich um ein metalogisches Symbol.
Eine besondere Bedeutung hat die **!, fail** - Kombination. Hiermit wird gezielt bestimmt, daß bei Benutzung einer bestimmten Klausel eines Prädikats, dieses falsch ist.
Bsp.: `verschieden(X,Y)` ist wahr, wenn X und Y verschieden sind, d.h. X und Y passen nicht zusammen ("matchen nicht"). In einer Regel könnte man das so formulieren: Wenn X und Y zusammen passen, dann scheitert verschieden(X,Y). In Prolog läßt sich diese Regel so ausdrücken:

```
verschieden(X,X) :- !, fail.
verschieden(X,Y).
```

In vielen Prolog-Dialekten gibt es das Prädikat „repeat“. Es läßt sich leicht implementieren, z.B.:

```
repeat.
repeat :- repeat.
```

„`repeat`“ kann unter keinen Umständen scheitern (in diesem Sinn ist es das genaue Gegenteil von `fail`). Mit diesem Prädikat kann man über Backtracking neue Lösungen angeben und damit den Programmfluß so lange anhalten, bis eine bestimmte Situation eintritt.

Negation in Prolog

Prolog-Klauseln, z.B.

```
b(....) :- a1(....), a2(.....).
b(....).
```

erlauben nur die Darstellung positiver Funktionen, d.h.: Prolog kann immer nur beweisen, daß etwas der Fall ist, nicht aber, daß etwas nicht der Fall ist. So ist bspw. ⊢ p(3) keine logische Folgerung aus dem folgenden Programm:

```
p(1).
p(2).
```

Prolog antwortet aber auf die Frage:

```
?-p(3).
```

mit NO und das ist keine logische Folgerung aus dem Programm. Da p(3) nicht aus dem vorliegenden Programm folgt, schließt man ¬ p(3) gilt. Es handelt sich hier aber um einen Vorgang, der die Negation über das Scheitern eines Prädikats realisiert (**negation as failure**). Zur Unterscheidung der **Negation as Failure** von der normalen logischen Negation. ¬ schreibt man in Prolog **not**.

Negation as Failure kann zu Problemen führen, wenn negierte Ziele Variable enthalten, z.B.:

```
?-X=2,not(X=1).
```

Die Antwort von Prolog ist: X=2.
Kehrt man die Reihenfolge der Ziele um

```
?-not(X=1),X=2.
```

, dann ist die Antwort: NO.
Logisch gesehen sind beide Anfragen identisch. Negation as Failure kann aber nicht zu korrekten Ergebnissen führen, wenn ungebundene Variable auftauchen. Negation as Failure kann nur dann sicher verwendet werden, wenn keine Variablen vorliegen, oder wenn Variable der negierten Ziele vorher ausreichend instanziiert werden. Eine solche Instanziierung kann der Programmierer, wie das vorliegende Beispiel zeigt, durch Umordnen der Ziele erreichen.

Disjunktion

Eine logische ODER-Verbindung wird durch das Semikolon „;“ gekennzeichnet, z.B.:

```
ist_student(X) :-
 ist_informatik_student(X) ;    /* Disjunktion */
 ist_mathematik_student(X).
```

Anstelle dieser Schreibweise ist es möglich, die Komponenten der ODER-Verbindung als alternative Regeln in der folgenden Form untereinander aufzuführen:

```
ist_student(X) :- ist_informatik_student(X).
ist_student(X) :- ist_mathematik_student(X).
```

Der Universalalgorithmus

Die spezielle Bedeutung von Prolog als KI-Sprache liegt in der Bereitstellung eines **universellen Algorithmus**. Konventionelle Programmierung benutzt für jedes spezifische Problem einen eigenen, angepaßten Algorithmus. Die Architektur der Rechner (von Neumann-Computer) ist für spezielle algorithmische Lösungswege besonders gut geeignet. Das führt zu der folgenden Vorgehensweise:

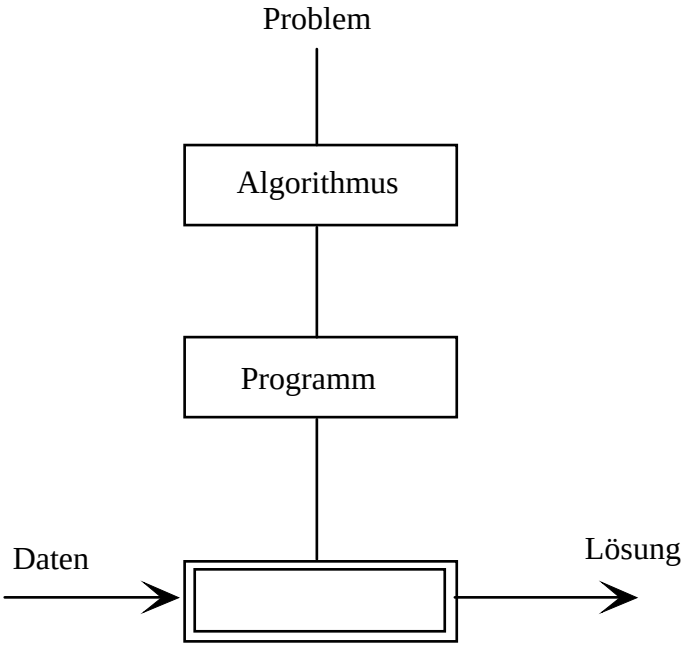


Abb. : Problemlösen mit angepaßtem Algorithmus

Die logische Programmierung schlägt einen anderen Weg ein. Sie versucht nur den Problembereich angemessen zu beschreiben, die eigentliche Problemlösung auf dem Rechner übernimmt ein universeller Algorithmus.

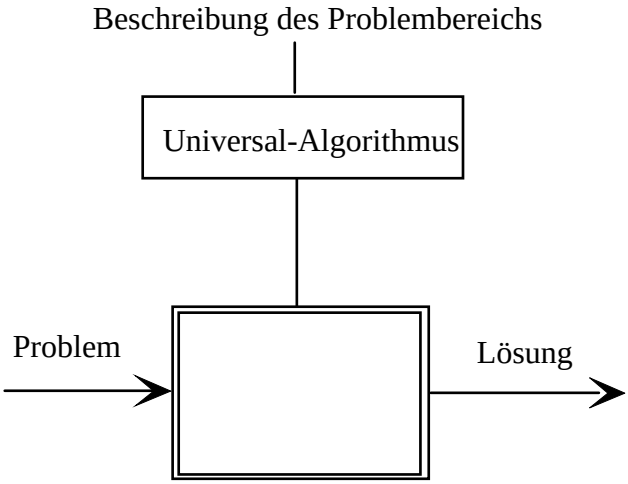


Abb. : Problemlösen mit Universal-Algorithmus

Man spricht in diesem Zusammenhang auch vom "nicht-algorithmischen Programmieren" und meint damit, daß der Programmierer nicht für die Konstruktion des universellen Algorithmus verantwortlich ist.

Der Prolog-Interpreter (der universelle Algorithmus) wertet bekanntlich die Wissensbasis (Sammlung von Fakten und Regeln) aus und benutzt dazu 3 grundsätzliche Verfahren: Resolution, Unifikation, Backtracking.
Grundsätzlich arbeitet Prolog folgendermaßen:

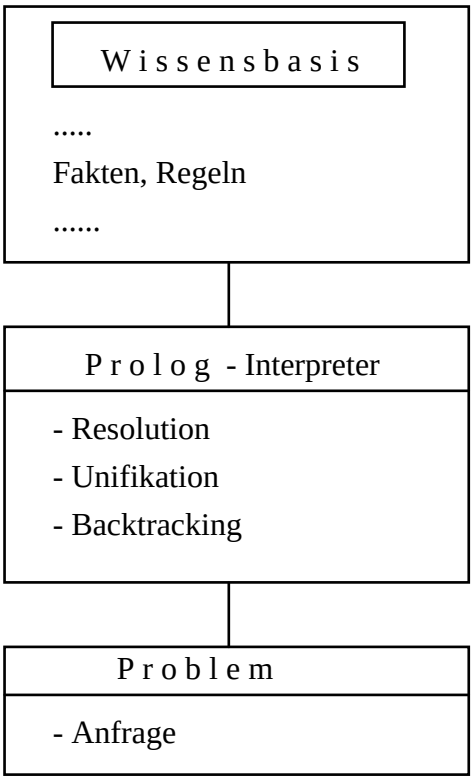


Abb. : Problemlösen mit Prolog

Zusammengesetzte Datenobjekte

Die einzige in Prolog verfügbare Beschreibungsmöglichkeit für Datenobjekte ist die des "Prolog-Terms". Ein **Term** ist entweder eine Konstante, Variable oder ein zusammengesetzter Term (bzw. zusammengesetztes Objekt).
Ein zusammengesetzter Term wird gebildet aus einem **Funktor** und Argumenten. Die Argumente eines zusammengesetzten Terms sind selbst wieder Terme.

Bsp.: `p(a,f(b),X)`
Argumente sind a, f(b) und X. "f" ist ein Funktor und bestimmt ein zusammengesetztes Objekt ("f(b)").

Eine **Struktur** ist ein zusammengesetztes Objekt, das aus mehreren anderen Objek-ten besteht.
Zusammengesetzte Objekte werden wie ein einzelnes Objekt behandelt, sie bestehen aus einem **Funktor** und den dazugehörigen Unterobjekten:

$\texttt{funktor(objekt}_1\texttt{,objekt}_2\texttt{,....,objekt}_N\texttt{)}$

Alle strukturierten Objekte können als Bäume dargestellt werden. Die Wurzel des Baumes ist der **Funktor**, und die Söhne der Wurzel sind die Komponenten. Ist

eine Komponente wiederum eine Struktur, so ist sie ein Teilbaum des Baumes der dem ganzen strukturierten Objekt entspricht.

Listen

Eine Liste ist eine spezielle Struktur in Prolog. Elemente einer Liste können Variable, Konstanten, Strukturen oder auch Listen sein. Sie sind in eckige Klammern eingeschlossen und durch Kommata getrennt:

```
[element_1, element_2, element_3, .... , element_n]
```

Die leere Liste wird so dargestellt: `[]`.

Listen können also 0 oder mehr Elemente enthalten. Eine leere Liste ist nicht weiter zerlegbar, nichtllere Listen sind über einen speziellen Operator (Symbol: **|**) in einen Listenkopf und einen Rest (der Liste) zerlegbar. Die Liste `[a,b,c,d]` wird durch `[a|b,c,d]` in a (Listenkopf, Head) und in die Liste `[b,c,d]` zerlegt. Listen sind **rekursive** Datenstrukturen: Der Typ "liste" wird durch sich selbst und definiert. Nur die Alternative (die Konstante `[]`) ist nicht selbstbezüglich. Rekursive Datenstrukturen lassen sich besonders einfach (in natürlicher Weise) mit **rekursiven Prädikaten** verarbeiten.
Bsp.: Das Aneinanderreihen 2er Listen ist in der Regel durch ein 3stelliges Prädikat „`verketten(liste_1,liste_2,Gesamtliste)`“ mit folgender Bedeutung definiert: Gesamtliste enthält nacheinander die Elemente von "liste_1" und "liste_2". Die Lösung ist zweckmässigerweise rekursiv:

```
verketten([],L,L).
verketten([Kopf|L1],L2,[Kopf|L3]) :- verketten(L1,L2,L3).
```

Der Aufruf "`verketten([a,b],[d,e,f],X)` führt zunächst in der vorliegenden Version von "verketten" zu "Kopf = a, L1 = [b], L2 = [d,e,f] und X = [a|L3]" und zu einem Aufruf "verketten([b],[d,e,f],L3]". Das ergibt "Kopf = b, L1 = [], L2 = [d,e,f], L3 = [b|L3]" und führt zum Aufruf "verketten([],[d,e,f],L3)". Mit der 1. Klausel erhält man L (bzw. L3) = [d,e,f] und das Ergebnis X = [a|[b|[d,e,f]]] = [a,b,d,e,f].

Mit Hilfe des Backtracking können nacheinander alle Objekte erzeugt werden, die irgendein Ziel erfüllen. Allerdings verschwindet eine Lösung (bzw. ist nicht mehr im Zugriff), wenn eine neue Lösung erzeugt wird. Manchmal ist es jedoch zweckmäßig, alle erzeugten Objekte in einer Liste zu sammeln. In Prolog gibt es dafür das Mengenprädikat **findall**.

```
findall(X,Z,L)
```

erzeugt die Liste aller Objekte X, für die das Ziel Z wahr ist.

Bsp.: Gegeben ist das folgende Prolog Programm

```
interesse(juergen,informatik).
interesse(herbert,mathematik).
interesse(wolfgang,statistik).
interesse(peter,linguistik).
interesse(josef,informatik).
```

Die Anfrage

```
?-findall(X,interesse(X,informatik),L)
```

ergibt

```
L=[juergen,josef]
```

Dynamische Datenbanken

Prolog ist flexibel: Fakten können während der Laufzeit hinzugefügt, gelöscht oder geändert werden. Das Hinzufügen, Löschen und Ändern von Daten übernehmen in Prolog eingebaute Prädikate. Beim Beenden der Arbeit mit dem Prolog-System geht der Inhalt der dynamischen Datenbank verloren, sofern er nicht gesichert wurde.
Zur Manipulation der dynamischen Datenbank gibt es Standarprädikate:

asserta
Damit werden der Datenbank im Arbeitsspeicher Fakten hinzugefügt. Der hinzu-fügende Fakt wird vor die bereits vorhandenen Klauseln gleichen Namens geschrieben. Durch `asserta(`*`praedikat`*`)` wird das als Argument aufgeführte Prädikat als Fakt in die aktuelle dynamische Datenbank an vorderster Stelle eingetragen.[128]

assertz
fügt einen Fakt hinzu. Der Fakt wird nach allen bereits vorhandenen Klauseln gleichen Namens geschrieben. Mit `assertz(`*`praedikat`*`)` wird das als Argument übergebene Prädikat als Fakt hinter die Klauseln der aktuellen Wissensbasis eingetragen. Der Fakt wird erst dann auf eine mögliche Unifizierung hin überprüft, wenn alle anderen Klauseln mit gleichem Prädikatsnamen zuvor als nicht unifizierbar erkannt sind.

retract
löscht Fakten aus der Datenbank. Durch den Aufruf von retract wird die 1. Klausel, die mit dem bei `retract` anzugebenden Argument unifizierbar ist, gelöscht. Mit `retract(`*`praedikat`*`)` wird der erste Fakt gezielt aus der Wissensbasis gelöscht, der mit dem Argument von retract unifiziert werden kann.

abolish
löscht sämtliche Klauseln eines Prädikats aus der Wissensbasis. Das Prädikat und die Stelligkeit des Prädikats, das aus der Wissensbasis entfernt werden soll, werden über `abolish(`*`praedikatsname, stelligkeit`*`)` angegegeben.

Sichern und Wiederherstellen

Zur langfristigen Sicherung der Fakten, die im dynamischen Teil der Wissensbasis enthalten sind, stehen die Prädkate `tell`, `listing` und `told` bereit:
Zum Zugriff auf den Inhalt einer Sicherungsdatei dient das Prädikat `consult('dateiname')` bzw. `reconsult('dateiname')`.
Zur Prüfung, ob eine Datei existiert und der gewünschte Zugriff auf diese Datei erlaubt ist, läßt sich das Standard-Prädikat `exists('dateiname',` *`art_des_zugriffs`*`)` einsetzen. Je nachdem, ob auf die Datei, deren Name im 1. Argument aufgeführt ist, zugegriffen wird, ist im 2. Argument „r“ oder „w“ (ohne Hochkommata) anzugeben.

[128] In vielen Prolog-Systemen können auch Regeln in den dynamischen Teil der Wissensbasis eingetragen weden. Dazu umfaßt das Prädikat „asserta“ zwei Argumente. Das 1. Argument enthält den Regelkopf, das zweite Argument den Regelrumpf.

2.2.2.4 Relationenkalkül und Prädikatenlogik

Ein Ausdruck des Relationenkalküls hat die Form: {t| p(t)}

t: Tupelvariable
p(t): spezielle Formel der Prädikatenlogik 1. Ordnung

Atome des Relationenkalküls sind:

- $t \in r$

t: Tupelvariable; r: Relation

Tupel enthalten Fakten. Eine Relation faßt Fakten zusammen. Eine relationale Datenbank kann damit umfassend logisch (Prädikatenlogik) interpretiert werden. Die Interpretation führt zur Wissensbasis.

- $t_1(A) \quad \Theta \; t_2(B)$

t_1, t_2: Tupelvariable; A,B: Attribute; $t_1(A)$ ist über A, $t_2(B)$ ist über B definiert

$t(A) \; \Theta \; k$

t(A): Tupelvariable t (über A definiert); $\Theta \in \{<, <=, =, >=, >, <>\}$

$k \in dom(A)$ (Konstante aus dem Wertebereich des Attributs A)

Formeln des Relationenkalküls sind folgendermaßen definiert:

- Ein Atom ist eine Formel
- Sind p und q Formeln, dann sind auch $\neg$ p, p $\wedge$ q, p $\vee$ q Formeln

- Ist p eine Formel und t eine freie (d.h. noch nicht durch einen Quantor gebundene) Tupelvariable, so sind auch "$\forall$t:p(t) und $\exists$t:p(t)" Formeln.

Wesentliche Unterschiede zur Prädikatenlogik 1.Ordnung sind:

- keine Funktionssymbole (Funktoren)
- nur Tupelvariable
- eingeschränkte Auswahl an Prädikaten ($\in$,Θ)

2.2.3 Datenbankmanipulationssprachen mit Bezugspunkten zur Relationenalgebra bzw. zum Relationenkalkül

Der Auswahlprozeß (Wiederauffinden, Modifizieren, Einfügen und Löschen) der Daten einer DB kann über eine Datenbankmanipulationssprache (DML) mit unterschiedlichen Sprachelementen vollzogen werden:

1. Auswählen mit Sprachelementen, die sich vorwiegend auf Elemente der Kontrollogik abstützen.
   Die Spezifizierung gewünschter Daten erfolgt durch Angabe der Operationenfolge, wie die Daten bereitzustellen sind ("**Wie**"-Sprachen, prozedurale DML)
2. Auswählen mit Sprachelementen von vorwiegend deskriptiver Art.
   Es ist nur das zu beschreiben, was als Ergebnis gewünscht wird. Also bspw. bestimmte Mengen von Sätzen (multiple records at a time logic) , die das Beschreibungsmerkmal erfüllen. Deskriptive Sprachen heißen auch "**Was**"-Sprachen.

Die allgemeinen Aufgaben dieser Sprachen bestehen in der Spezifizierung gewünschter Tabellen oder von Mengen aus vorhandenen Relationen. Aus den Relationen der DB sucht man bestimmte in einer Abfrage spezifizierte Tupel auf und stellt sie für eine Auswertung zusammen. Die Tupel dürfen dabei aus verschiede-nen Relationen stammen (Benutzersätze der DB).
Man unterscheidet die Datenbankmanipulationssprachen (DML) relationaler Datenbanken in

- Prädikatenkalkülsprachen

Hier wird die Ergebnisrelation beschrieben durch Angabe der Eigenschaften (Prädikate), die die Relationen haben müssen. Weiterhin unterscheidet man zwischen tupelorientierten und domänenorientierten Sprachen, je nachdem, ob die Variablen, die in den Formeln des Kalküls vorkommen, für Tupel oder für Komponenten der Tupel (Werte aus dem Wertebereich) stehen.

- Algebraische Sprachen

Hier wird die Ergebnisrelation beschrieben als Resultat von Mengenoperationen der Relationen der Datenbank.

Weitgehend stützen sich die Datenmanipulationssprachen für relationale Datenbanken auf das **Prädikatenkalkül** ab. Der erste Vertreter dieser Sprachen war die bereits von Codd vorgeschlagene Sprache ALPHA[129]. Verbreitete Implementierungen für relationale Datenbanken sind:

- das standardisierte **SQL** (Structure Query Language)
- **QBE**-Sprachen (Query by Example)

[129] vgl. Wedekind, Hartmut: "Datenbanksysteme I", Mannheim/Wien/Zürich, 1974

2.3 SQL

SQL[130] (Structured Query Language) ist im wesentlichen eine tupelorientierte Prädikatenkalkülsprache, enthält aber auch algebraische Elemente. Die zentrale Idee, die SQL zugrundeliegt, ist eine Abbildung zwischen bestimmten Spalten einer Tabelle. Die Tabelle (bzw. die in die Tabelle aufgenommenen Daten) bilden den Definitionsbereich. Über ein Auswahlkriterium wird aus dem Definitionsbereich ein Abbildungsbereich bestimmt. SQL (Structured English Language) drückt das so aus:

```
select B                    /* Bildbereich */
from   R1                   /* Definitionsbereich */
where  A in                 /* Auswahlkriterium */
(select A                   /* Bildbereich */
 from   R2                  /* Definitionsbereich */
 where  K = 'Konstante');
```

Die Auswertung eines solchen Ausdrucks erfolgt von oben nach unten. Zuerst wird ein "**select-from-where**"-Block (Grundbaustein der Sprache SQL) spezifiziert, der mit Daten gespeichert wird, die von einem unteren Block geliefert werden. Der untere Block wird mit Daten gefüllt, die aus Selektionen resultieren, der obere Block führt über diese Daten eine Projektion aus.

2.3.1 SQL/92

Übersicht

Bisher wurden von ANSI, ISO zur Standardisierung von SQL herausgegeben:

SQL/86 (SQL-Sprachstandard, der im Jahre 1986 verabschiedet wurde)
SQL/89 (auch ANSI-89 genannt)
SQL/92 (auch SQL2 genannt)

SQL2 kennt drei Standardisierungsebenen: **Entry-SQL**, **Intermediate-SQL** und **Full-Level-SQL**. Der Sprachumfang einer unteren Ebene (Level) ist immer eine echte Teilmenge der nächsthöheren Ebene. Entry-SQL entspricht den ANSI-89-Standard mit minimalen Erweiterungen, Intermediate-SQL kennt schon eine Vielzahl von Erweiterungen, Full-Level-SQL realisiert den Standard vollständig.
Full-Level-SQL unterscheidet datendefinierende (DDL) und datenmanipulierende (DML) Sprachelemente. Die DDL erlaubt neben der Strukturdefinition von Tabellen: Festlegung von Integritätsbedingungen, Schlüsselkandidaten, Primär-, Femdschlüssel, Spalten- und Tabellenbedingungen, generelle Versicherungen (Assertions), etc.
Die DML enthält Operationen zur mengenorientierten Abfrage und Manipulationen der Daten. SQL/92 bietet mehrere neue `JOIN`-Operatoren an sowie die Mengenoperationen: Vereinigung, Schnittmenge, Differenz.

[130] SQL wurde im Rahmen eines IBM-Forschungsauftrages Mitte der siebziger Jahre entwickelt. Seit der Markteinführung von SQL im Jahre 1979 haben zahlreiche Unternehmen SQL als Standard-Datenbanksprache für Großrechner und auch für Minicomputer übernommen.

Im wesentlichen umfassen die Erweiterungen zu SQL/89 Datenbankabfragen (insbesondere den JOIN-Befehl) und Methoden zur Abbildung von Integritätsbedingungen.

Datenbankabfragen

Join-Befehl

SQL/89 war kein expliziter **JOIN**-Befehl bekannt. Der **JOIN** wurde implizit durch die Angabe mehrerer Tabellen und einer Verknüpfungsopration in der WHERE-Klausel hergestellt[131]. **SQL/92** bietet den **JOIN**-Befehl in mehreren Varianten an: Cross-, Inner- Outer- (Left-, Right-, Full-Outer-JOIN) und Union-Join.

Cross-JOIN
bildet das kartesische Produkt zweier Tabellen, d.h.: Jeder Datensatz der einen Tabelle wird mit jedem Datensatz der anderen Tabelle verknüpft, z.B.:

```
SELECT * FROM ANGESTELLTE
 CROSS JOIN ABTEILUNG;
```

Die zum Cross-JOIN äquivalente Anweisung ist:

```
SELECT * FROM ANGESTELLTE, ABTEILUNG;
```

Natural-JOIN
verknüpft die beiden angegebenen Tabellen über die Gleichheit aller gleichlautenden Spaltennamen. In der Ergebnismenge werden gleichlautende Spaltennamen nur einmal angezeigt. Haben die beiden Tabellen keine Spalten gemeinsam, so degeneriert der Natural-Join zum Cross-Join.

Inner-JOIN
verknüpft 2 Tabellen durch eine Bedingung. Das Schlüsselwort „Inner“ muß nicht angegeben werden. Der Inner-JOIN existiert in 2 Varianten.

In der 1. Variante wird die Verknüpfungsbedingung in einer ON-Klausel angegeben:

```
SELECT * FROM Angestellte, Abteilung
 Angestellte INNER JOIN Abteilung
 ON Angestellte.Abt_ID = Abteilung.Abt_ID;
```

In der 2. Variante kann man durch die USING-Klausel beliebig viele, für beide Tabellen gleichlautende Feldnamen angeben. Der JOIN wird dann über gleiche Werte dieser Felder gebildet.

```
SELECT * FROM Angestellte, Abteilung
 Angestellte INNER JOIN Abteilung
 USING Abt_ID;
```

Äquivalent dazu ist in SQL/89

[131] Vgl. 1.4.3.2

```
SELECT * FROM Angestellte, Abteilung
WHERE Angestellte.Abt_ID = Abteilung.Abt_ID;
```

Der „Inner-“JOIN nimmt nur Datensätze einer Tabelle in die Ergebnismenge auf. Der „Outer-JOIN“ dagegen gewährleistet: Bei der JOIN-Verknüpfung erscheint jeder Datensatz der Ausgangstabelle in der Ergebnistabelle, auch wenn er kein Pendant in der verknüpften Tabelle hat. Das erfolgt beim „Left Outer JOIN“ für die linke Tabelle, beim „Right Outer JOIN“ für die rechte und beim „Full Outer JOIN“ für beide.
Der Outer JOIN ist bei der Verknüpfung von Tabellen über Primär- und Fremdschlüssel von Bedeutung, falls dabei der Fremdschlüssel den Nullwert annehmen darf. Auch vom Outer JOIN existieren 2 Varianten, eine mit ON und eine mit USING-Klausel.

Union JOIN
Alle Datensätze der ersten und zweiten Tabelle werden in die Ergebtabelle mit aufgenommen. Im Gegensatz zum OUTER JOIN wird aber nicht versucht, die Datensätze über eine Bedingung oder über die Datenfelder miteinander zu verknüpfen.

Mengenoperationen

Der UNION-Operator bildet die Vereinigungsmenge, der INTERSECT-Operator die Schnittmenge und der EXCEPT-Operator die Differenz aus beiden Tabellen. Doppelte Datensätze werden bei allen 3 Operatoren aus der Ergebnismenge eliminiert. Alle Befehle gibt es in 3 verschiedenen Formen, die am UNION-Operator demonstriert werden soll:

```
SELECT * FROM Tabelle1
UNION
SELECT * FROM Tabelle2
```

Der UNION-Operator setzt voraus: Beide Tabellen verfügen über die gleiche Anzahl Felder und korrespondierende Spalten besitzen korrespondierende Datentypen. Korrespondieren 2 Tabellen nicht vollständig miteinander, dann kann man sie über ausgewählte Felder vereinigen. Der Vereinigung geht dann eine Projektion bzgl. der angegebenen Felder voraus.

```
SELECT * FROM Tabelle1
UNION CORRESPONDING BY (Feld1, Feld2)
SELECT * FROM Tabelle2
```

Die 3. Variante ermöglicht eine implizite Auswahl korrespondierender Felder über Namensgleichheit:

```
SELECT * FROM Tabelle1
UNION CORRESPONDING
SELECT * FROM Tabelle2
```

Methoden zur Abbildung von Integritätsbedingungen

Tabellen- und Spaltenbedingungen

Tabellenbedingung: Das ist ein bedingter Ausdruck (**Table Constraint**), der in einer Tabellendefinition angegeben wird und sich auf mehrere Spalten der gleichen oder auch anderer Tabellen beziehen kann.
Spaltenbedingung: Eine derartige Bedingung (**Column Constraint**) bezieht sich nur auf eine Spalte der Tabelle. Sie kann unmittelbar in der Spaltendefinition oder als eigenständiges Element der Tabellendefinition angegeben werden. Die beiden einfachsten Spaltenbedingungen sind NOT NULL[132] und UNIQUE[133].
Tabellen-, Spaltenbedingungen können in der CHECK-Klausel[134] definiert werden.

Bsp.: „Anrede“ unterliegt der Spaltenbedingung `CHECK (VALUE IN ('Herr','Frau'))`.

Diese Integritätsregel ist aber genauer betrachtet keine spezifische Eigenschaft der Spalte „Angestelle.Anrede“ sondern eine Regel, die generell für den Wertebereich von ‘Anrede’ Gültigkeit besitzt.
Über das **Domänen**konzept können in SQL/92 Wertebereiche unabhängig von Datenbankfeldern modelliert werden. Ein **Domain** besitzt einen Namen und wird auf einen Datentyp abgebildet. Zusätzlich lassen sich für Domains Integritätsbedingungen und Defaultwerte definieren. Die Domain kann bereits im CREATE-TABLE-Statement zur Angabe eines Datentyps verwendet werden, z.B.:

```
CREATE DOMAIN Anreden
AS CHAR(4) CHECK VALUE IN ('Herr','Frau'))

CHECK TABLE Angestellte (
........
 Anrede Anreden
....... )
```

Für alle Wertebereiche, die in der Datenbank mehrfach verwendet werden, sind Domains geeignet. Änderungen von Datentypen, z.B. von Primärschlüsseln, auf die eine Menge anderer Fremdschlüssel verweisen, lassen sich fehlerfrei (und konsistent) durchführen.

Assertions (generelle Versicherungen)

Sie sind nicht mit einer Spalte oder eine Tabelle verknüpt. Die Defintion erfolgt direkt über CREATE ASSERTION ..., z.B.:

```
CREATE ASSERTION AbtGroesse
 CHECK (NOT EXIST
 (SELECT * FROM Abteilung
  WHERE (SELECT COUNT(*) FROM Angestellte
   WHERE       Angestellte.Abt_ID       =       Abteilung.Abt_ID)       >
Abteilung.MaxAngstellte))
```

Eine Assertion erhält einen Namen (z.B. „AbtGroesse) und ist ein bedingter Ausdruck, der eine Integritätsbedingung formuliert. Eine Assertion ist (im Gegensatz zum Constraint) weder einer Spalte noch einer Tabelle zugeordnet.

---

[132] das Datenfeld muß immer nur einen Wert enthalten

[133] legt eine Spalte fest: Mit der CHECK-Klausel werden Tabelle, Spaltenbedingung definiert

[134] Die besteht aus einem bedingten Ausdruck, der mehrere miteinander durch AND, OR oder NOT verknüpfte Prädikate verwenden darf

Relationale Integrität

Eine relationale Datenbank muß sich an die durch das relationale Modell bedingten Integritätsregeln der Primär- und Fremdschlüsselintegrität (Entity- und Referential Integrity) halten.
„**Entity Integrity**“ besagt: Jede Tabelle muß ein Attribut oder eine Attributmenge besitzen, die Datensätze eindeutig identifiziert. „**Referential Integrity**“ bezieht sich auf die Abbildung von Beziehungen durch Fremdschlüssel. jeder Fremdschlüssel muß auf einen existierenden Primärschlüssel referenzieren.

Bsp.:

```
CREATE TABLE Abteilung (
 ID CHAR(2),
 Bez CHAR(40),
 MaxAngestellte INTEGER,
 PRIMARY KEY(ID))

Create TABLE Angestellte (
 ID CHAR(3) NOT NULL,
....
 ABT_ID CHAR(2),
 PRIMARY KEY(ID)
 FOREIGN KEY(Abt_ID) REFERENCES Abteilung[(ID)]
```

In der **FOREIGN KEY** - Klausel wird die Referenz auf die Tabelle Abteilung definiert. Eine Änderung des Primärschlüssels verletzt in der referenzierenden Tabelle die referentielle Integrität. Mit referentiellen Aktionen (referential actions) legt man während der Fremdschlüsseldefinition fest, wie die Datenbank auf das Ändern (ON UPDATE) oder Löschen (ON DELETE) eines Primärschlüssels reagieren soll:

CASCADES
wird der Primärschlüssel geändert / gelöscht, so werden alle referenzierten Datensätze ebenfalls geändert / gelöscht.

NO ACTION
Der Primärschlüssel wird ohne weitere Prüfung geändert / gelöscht. Sollte dadurch die referentielle Integrität verletzt werden, dann zieht das System die Operation zurück. Ändern oder Löschen hat nur Erfolg, wenn kein Datensatz auf diesen Primärschlüssel referenziert.

SET NULL
Alle Fremdschlüssel in den referenzierenden Datensätzen erhalten den Wert Null. Diese Aktion ist natürlich nur dann möglich, falls sie entsprechenden Felder nicht mit dem Column-Constraint NOT NULL belegt sind.

SET DEFAULT
Alle Fremdschlüssel werden mit ihrem Defaultwert belegt. Ist kein DEFAULT definiert, wird der Nullwert zugewiesen.

Zeitpunkt der Prüfung

Es kann ein Zeitpunkt für Integritätsprüfungen angegeben werden. Jede Integritätsbedingung ist zu jedem Zeitpunkt eine Transaktion in einem der 2 Modi: IMMEDIATE oder DEFERRED.
IMMEDIATE bedeutet: Diese Bedingung wird nach jeder Ausführung einer SQL-Anweisung überprüft.
DEFERRED legt fest: Die überprüfung erfolgt erst am Ende einer Transaktion.
SET CONSTRAINT IMMEDIATE / DEFERRED legt den Modus fest. Bereits bei der Defintion einer Integritätsbedingung kann bestimmt werden, in welchem Modus sie arbeiten soll (INITIALLY IMMEDIATE / INITIALLY DEFERRED), bzw. ob sie überhaupt verzögert ausgeführt werden kann (DEFERABLE / NOT DEFERABLE). Ohne zusätzliche Angaben gilt defaultmäßig NOT DEFERABLE.

Operationale Integrität

Sie schützt im Mehrbenutzerbetrieb bei konkurrierendem Zugriff vor Inkonsistenzen. Die Erhaltung der operationalen Integrität wird in einer datenbank durch Transaktionen gewährleistet. Der Beginn der Transaktion wird implizit mit bestimmten SQL-Anweisungen ausgelöst. Eine Transaktion kann damit mit COMMIT ordnungsgemäß zurückgesetzt werden. Mit ROLLBACK lassen sich Änderungen seit Beginn der Transaktion rückgängig machen.

```
COMMIT [WORK]
ROLLBACK[WORK]
SET TRANSACTION [READ ONLY | READ WRITE]

 [ISOLATION LEVEL { READ UNCOMITTED | READ COMMITTED
    | REPEATABLE READ | SERIALIZABLE } ]
```

Isolation Levels

Zur Synchronisation des Mehrbenutzerbetriebs bietet SQL/92 über SET TRANSACTION vier abgestufte Isolationsebenen (ISOLATION LEVEL) an. Die unterschiedlichen „Isolation Levels“ lassen unterschiedliche Effekte bei der Transaktionsverarbeitung zu bzw. schliessen sie aus. Jeder Isolation Level schließt bestimmte Phänomene, die im Mehrbenutzerbetrieb auftreten können, aus bzw. nimmt sie in Kauf. In konkurrierenden Transaktionen können folgende Phänomene auftreten:

Lost Update
Ein `Lost Update` in einer Transaktion ist eine Veränderung in dieser Transaktion, die von einer anderen Transaktion überschrieben wird.
Dirty Read
Ein `Dirty Read` in einer Transaktion ist ein Lesevorgang mit Primärschlüssel, der veränderte Zeilen anderer noch nicht terminierter Transaktionen liest. Es können sogar Zeilen gelesen werden, die nicht existieren oder nie existiert haben. Die Transaktion sieht einen temporären Schnappschuß der Datenbank, der zwar aktuell ist, aber bereits inkonsistent sein kann. Offensichlich erfaßt die Abfrage Ergebnisse einer nicht (mit `COMMIT`) bestätigten Transaktion und erzeugt dadurch ein falsches Ergebnis.
Non-Repeatable Read
Ein `Non-Repetable Read` in einer Transaktion ist ein Lesevorgang mit Primärschlüssel, der im Falle von mehrmaligem Lesen zu unterschiedlichen Ergebnissen führt. Innerhalb einer Transaktion führt die mehrfache Ausführung einer Abfrage zu unterschiedlichen Ergebnissen, die durch zwischenzeitliche Änderungen (`update`) und Löschungen (`delete`) entstehen. Die einzelnen Abfragergebnisse sind konsistent, beziehen sich aber auf unterschiedliche Zeitpunkte.
Phantom-Read

Ein Phantom einer Transaktion ist ein Lesevorgang bzgl. einer Datenmenge, die einer bestimmten Bedingung genügen. Fügt eine andere Transaktion einen Datensatz ein, der ebenfalls diese Bedingung erfüllt, dann führt die Wiederholung der Abfrage innerhalb einer Tarnsaktion zu unterschiedlichen Ergebnissen. Bei jeder Wiederholung einer Abfrage innerhalb einer Transaktion enthält das zweite Abfragergebnis mehr Datensätze als das erste, wenn in der Zwischenzeit neue Datensätze eingefügt wurden.

SQL/92 definiert vier „Isolation Level: Read Uncommitted, Read Comitted, Repeatable Read, Serializable“, die nur bestimmte der angegebenen Effekte zulassen bzw. ausschließen.

Effekte: / Isolation Level	Lost Update	Dirty Read	Non-Repeatable Read	Phantom
0: Read Uncommitted	nicht erlaubt	erlaubt	erlaubt	erlaubt
1: Read Committed	nicht erlaubt	nicht erlaubt	erlaubt	erlaubt
2:Repeatable Read	nicht erlaubt	nicht erlaubt	nicht erlaubt	erlaubt
3: Serializable	nicht erlaubt	nicht erlaubt	nicht erlaubt	nicht erl.

Abb. 2.3-1:

`Dirty Read` läßt im Isolation Level zu: Die eigene Transaktion darf Daten lesen, die andere Transaktionen geändert haben, obwohl die anderen Transaktionen nicht mit COMMIT beendet wurden. Im Fall `Read-Committed` ist das ausgeschlossen. Hier kann aber beim `Non-Repeatable Read` vorkommen: Die Datensätze , die die eine Transaktion liest, werden vor Ende dieser Transaktion durch andere Transaktionen verändert. Dadurch kann wiederholtes Lesen desselben Datensatzes innerhalb einer Transaktion unterschiedliche Ergebnisse liefern. `Repeatable Read` verhindert diesen Effekt. Das Phantomproblem verhindert der Isolation Level SERIALIZABLE.

Ausgewählte Befehle aus dem Full-Level-SQL

Datendefinitionen (DDL)

Schema- und Tabellendefinition
Ausgangspunkt ist das Schema, das Datenbeschreibungen zu Domains, Tabellen, Views, Integritätsbedingungen und Zugrifsrechten umfaßt:

```
CREATE SCHEMA [schema] [schema-element-list]
```

Schema-Element:

```
   domain-definition
|  base-table-definition
|  view-definition
|  authorization-definition
|  general-constraint-definition
```

Domain-Defintion:

```
CREATE DOMAIN domain [AS] data-type
 [DEFAULT { literal | NULL } ]
 [[CONSTRAINT constraint] CHECK (conditional-expression)
   [INITIALLY {DEFERRED | IMMEDIATE} ]
```

```
   [ [NOT] DEFERABLE]]
```

Base-Table-Definition:

```
CREATE [ TEMPORARY ] TABLE basetable
 {column-definition-list | base-table-constraint-definition-list )
  [on Commit { DELETE | PRESERVE } ROWS ]
```

Mit der ON COMMIT-Klausel wird festgelegt, ob bei jedem COMMIT alle Datensätze der Tabelle gelöscht werden oder erhalten bleiben. Temporäre Tabellen eignen sich ideal zum Speichern von Zwischenergebnissen.

```
CREATE VIEW [(column-commalist)]
 AS table-expression [WITH [CASCADED | LOCAL ] CHECK OPTION[135] ]
```

Authorization-Definition

```
GRANT {privilege-commalist | ALL PRIVILEGES}
 ON {DOMAIN domain | [TABLE] table}
 TO { user [, user + ] | PUBLIC }
```

Mit dem GRANT-Befehl erfolgt die Definition der Zugriffsrechte für Tabellen und Domains. REVOKE nimmt die mit GRANT vergegebenen Rechte zurück. RESTRICT / CASCADE bezieht sich hier auf die der GRANT-Option weitervergebenen Rechte:

Privilege:

```
 SELECT
| INSERT
| UPDATE
| DELETE
| REFERENCES
| USAGE

REVOKE [ GRANT OPTION FOR ] privelege-commalist
ON { DOMAIN domain | [TABLE] table }
FROM { user [,user +] | PUBLIC }{RESTRICT | CASCADE}
```

Column-Defintion:

```
column  {data-type | domain}
 [DEFAULT {literal | NULL}]
 [column-constraint-definition]
```

Data Types

CHARACTER [n]	Zeichenkette fester Länge n, Abkürzungen: CHAR[n] und CHAR für CHAR[1]
CHARACTER VARYING [n]	Zeichenkette variabler Länge, maximal n, abgekürzt VARCHAR
BIT [n]	Bitfolge fester Länge n
BIT VARYING [n]	Bitfolge variabler Länge, maximal n

[135] nur relevant, wenn der View updatefähig ist.

NUMERIC (p,q)	Dezimalzahl mit genau p Vorkomma- und q Nachkommastellen, NUMERIC(p) wenn ohne Nachkommastellen
DECIMAL (p,q)	Dezimalzahl mit mindestens p Vorkomma- und q Nachkommastellen, DECIMSAL(p) wenn ohne Nachkommastellen
INTEGER	Ganze Zahl mit Vorzeichen. Die Größe ist ebenfalls der Implementierung überlassen.
SMALLINT	Ganze Zahl mit Vorzeichen. Die Größe ist ebenfalls der Implementierung überlassen, darf aber die von INT nicht überschreiten.
FLOAT(p)	Fließkommazahl mit p Nachkommastellen, alternative Bezeichnung ist REAL
DATE	Datum
TIME	Zeit
TIMESTAMP	Zeitstempel

Abb.: Datentypen in SQL/92

Änderungen der Datendefinitionen können über den Befehl ALTER ausgeführt werden:

```
ALTER DOMAIN domain
 SET DEFAULT {literal | NULL }
| DROP DEFAULT
| ADD [CONSTRAINT constraint] CHECK (conditional-expression)
  [INITIALLLY [DEFERRED | IMMEDIATE} ]
  [ [NOT] DEFERABLE]]
| DROP constraint

ALTER TABLE base-table
 ADD [column] column-definition
| ALTER [ COLUMN ] column { SET DEFAULT { literal | NULL}
| DROP DEFAULT
| DROP [ COLUMN ] column { RESTRICTED | CASCADE }
| ADD base-table-constraint-definition
| DROP CONSTRAINT constraint {RESTRICTED | CASCADES }

DROP SCHEMA schema { RESTRICT | CASCADE }
DROP DOMAIN domain { RESTRICT | CASCADE }
DROP TABLE base-table [ RESTRICT | CASCADE }
DROP VIEW view { RESTRICT | CASCADE }
DROP ASSERTION constraint
```

Die Optionen RESTRICT / CASCADE im DROP-Befehl bestimmen das Verhalten der Operation, falls das zu entfernende Objekt (Spalte, Tabelle, CONSTRAINT, +) noch in anderen Defintionen verwendet wird. RESTRICT verbietet das Löschen, CASCADE löscht zusätzlich alle Definitionen, die dieses Objekt ebenfalls verwenden.

Base-Table-Constraint-Defintion:

```
[CONSTRAINT constraint]
[PRIMARY KEY | UNIQUE } ( column-commalist )
 [INITIALLY { DEFERRED | IMMEDIATE } ]
 [ NOT ] DEFEABLE ]]
|   FOREIGN   KEY   (column-commalist)   REFERENCES   base-table   [column-
commalist)]
 [ MATCH { FULL | PARTIAL}]
 [ ON DELETE { NO ACTION | CASCADE | SET DEFAULT | SET NULL } ]
```

```
 [ ON UPDATE { NO ACTION | CASCADE | SET DEFAULT | SET NULL }]
 [ INITIALLY { DEFERRED | IMMEDIATE }]
 [[ NOT ] DEFERABLE ]]
| CHECK conditional-expression
 [ INITIALLY { DEFERD | IMMEDIATE} ]
 [ [ NOT ] DEFERABLE ]]
```

Die Option MATCH PARTIAL erlaubt, daß in zusammengesetzten Fremdschlüsseln einzelne Spalten den Null-Wert annehmen. Die Referenz muß nur für die Spalten erfüllt sein, die nicht Null sind.

Column-Constraint-Defintion:

```
NOT NULL
[INITIALLY {DEFERRED | IMMEDIATE } ]
[[NOT] DEFERRABLE]]
| { PRIMARY KEY | UNIQUE } ]
 [ INITALLY { DEFERRED | IMMEDIATE } ]
 [[NOT] REFRABLE ]]
| REFERENCES base-table [ (column)]
 [ ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL }]
 [ INITIALLY {DEFERED | IMMEDIATE}]
 [[NOT] DEFERABLE ]]
| CHECK conditional-expression
 [ INITIALLY { DEFERRED | IMMEDIATE } ]
 [ [NOT] DEFERABLE ]]
SET CONSTRAINT (constraint-commalist | ALL { DEFERRED | IMMEDIATE }
```

2.3.2 Oracle-SQL

2.3.2.1 Das Datenbanksystem Oracle

Oracle ist ein Produkt der Oracle Corporation, Belmont, California, USA. Oracle[136] wird hauptsächlich auf UNIX-Rechnern eingesetzt. Es handelt sich um ein relationales DB-System, das über **SQL** leicht zugänglich ist.

2.3.2.1.1 Speichern und Verwalten von Daten

Gemeinsamer Speicher für die durch SQL-Anweisungen ausgelösten Datenbankoperationen ist die **System Global Area** (**SGA**). Sie enthält Datenbank- und Logpuffer sowie Informationen über das **Data Dictionary** (**DD**). Die **SGA** ist der Hauptspeicherbereich, auf den alle Benutzer einer Datenbank zugreifen und besteht aus dem Blockpuffer, dem Logpuffer, dem Shared Pool und einem fest allokierten Bereich. Der Blockpuffer bietet Platz für die beim Systemstart festgelegte Anzahl von Datenbank-Blöcken. Im Logpuffer werden für die Rekonstruktion der DB in Fehlerfällen Veränderungen an dem Datenbankpuffer protokolliert, bevor sie permanent gesichert werden. Der Shared Pool nimmt hauptsächlich SQL- und PL/SQL-Befehle sowie den Dictionary Cache zur Beschleunigung der Befehle auf. Der Dictionary Cache enthält Informationen aus dem Data Dictionary, die bei der Interpretation der Befehle benötigt werden.
Logisch gesehen besteht die Datenstruktur des Oracle-DBMS aus folgenden Sichten:

Segment
Beim Anlegen einer Tabelle wird automatisch ein bestimmter Speicherplatz für Daten und Indexe reserviert. Jede Tabelle hat genau ein Daten-Segment für die Daten der Tabelle und je Index ein Index-Segment. Der Oracle7Server kennt neben den Benutzer- oder Datensegmenten (`table`, `index`, `cluster`, usw.) für den Betrieb noch Rollback-Segmente zur Implementierung des Transaktionskonzepts (Speicher für Dateninhalte vor Datenänderung, sog. `Before Image` – Information) und temporäre Segmente für Sortieroperationen und damit verwandte Operationen (z.B. Gruppierungen). Wird bspw. eine große Menge an Speicherplatz dafür angefordert, dann wird auf temporäre Segmente (Speicherplatz für Zwischen-ergebnisse und das Resultat) ausgewichen. Nach erforderlicher Durchführung der Sortierung und Gruppierung werden temporäre Segmente wieder gelöscht.
Tablespace
Alle Segmente werden in Tablespaces (logische Einheiten zur Speicherzuteilung) angelegt. Ein Tablespace umfaßt mindestens eine Datenbankdatei (Tabelle) und stellt den datenbankdateien physisch Speicherplatz zur Verfügung. Eine Datenbank besteht immer aus mindestens einer Tablespace (System Tablespace), die aus mindestens einer physischen Datei besteht. Tablespaces können mit `CREATE TABLESPACE ...` erzeugt werden. Datenbankobjekte, z.B. Tabellen, Indexe, Cluster werden hier angelegt. Werden Tablespaces beim Anlegen von Objekten nicht explizit angegeben, wird der SYSTEM-Tablespace oder der für den jeweiligen Benutzer voreingestellte Tablespace zugeordnet. Struktur, Status und Größe eines `Tablespace` kann durch den Datenbank-Administrator (DBA) mit dem Befehl `ALTER TABLESPACE ...` geändert werden. Zwei widersprüchliche Zielvorstellungen beeinflussen den Aufbau von Tablespaces:

1. Daten eines Segments sollen physisch zusammenhängen.
2. Ein Segment soll dynamisch wachsen können

[136] Oracle Version 7

Extent
Diese Ziele werden über die Extent-Verwaltung optimiert. Jedes Segment wird unabhängig vom Segment-Typ als eine Folge von einem oder mehreren Extents verwaltet. Das ist die kleinste Einheit bei der Reservierung von Speicherplatz. Ein Extent besteht aus einem Block oder mehreren Oracle-Blöcken.
Database
Eine Database besteht aus einem oder mehreren Speicherbereichen (Tablespace). Ein DBMS kann mehrere Datenbanken verwalten.

Physisch gesehen besteht die Datenbank aus:

Datenbankdateien
Ein Tablespace umfaßt mehreren physische Datenbankdateien (z.B. Tabellen). Neben Datendateien existieren in einer Oracle-Datenbank noch weitere Dateien mit speziellen Aufgaben:

- Redo-Log-Dateien und Redo-Log-Gruppen
  Redo-Log-Dateien umfassen Protokolle zu den Änderungen an den Datenbank-Blöcken.
- Control-Dateien
  Sie enthalten neben Zeitstempeln, Konsistenzinformationen Angaben zur physischen Struktur der Oracle-Datenbank, d.h. Namen und Größenangaben aller Daten- und Online-Redo-Log-Dateien.
- `init.ora`-Datei
  Diese Datei enthält Initialisierungsparameter zur Bestimmung von Konfiguration und Verhalten der Oracle-Datenbank (Oracle7Server).

Blöcke
Jede Datei besteht aus einer Anzahl von Oracle-Blöcken. Jeder Oracle-Block bestehet aus einem oder mehreren Plattenblöcken.

2.3.2.1.2 Benutzer, Rollen und Berechtigungen

ORACLE-Benutzer können vom Datenbank-Administrator bestimmt werden und folgende Zugriffsrechte erhalten:

CONNECT
Ein mit dem CONNECT-Privileg ausgestatteter Benutzer hat folgende Rechte :

- Einloggen in das ORACLE-System
- Ansehen von Daten anderer Benutzer, falls dies von diesen Benutzern erlaubt wurde
- Durchführen von Daten-Manipulation-Kommandos auf Tabellen anderer Benutzer, falls dies von diesen Benutzern erlaubt wurde
- Erzeugen von Views und Synonymen

Nicht erlaubt ist

- Tabellen, Indexe oder Cluster einzurichten oder zu löschen
- Strukturen vorhandener Tabellen zu verändern

RESOURCE
Zusätzlich zu den CONNECT-Privilegien werden folgende Rechte erteilt :

- Erzeugen von Tabellen, Indexen und Cluster
- Zugriffsrechte auf diese Tabellen für andere Benutzer erteilen oder sperren
- Zugriffskontrolle über das AUDIT-Kommando auf alle eigenen Tabellen

Nicht erlaubt ist

- Tabellen manipulieren, die andere Benutzer eingerichtet haben (Ausnahme: Es liegt dafür eine Erlaubnis vor)
- anderen Benutzern das CONNECT- oder RESOURCE-Privileg zu erteilen oder zu entziehen

DBA
Zusätzlich zu den CONNECT- und RESOURCE-Rechten werden folgende Rechte erteilt :

- Zugriff auf alle Daten aller Benutzer und Anwednung aller SQL-Kommandos auf diese Daten
- Zuteilen und Sperren von Datenbank-Rechten für alle Anwender
- Synomyme für alle Anwender erzeugen
- Erzeugen und Ändern von Partitions

Das Zugriffsrecht DBA ermöglicht die Datenbank-Administration. Normalerweise ist mit diesen Rechten der Datenbankadministrator ausgestattet. Für die Vergabe von Zugriffsrechten steht die GRANT TO-Anweisung bereit:

```
GRANT {CONNECT|RESOURCE|DBA}
TO Benutzerliste
IDENTIFIED BY Paßwortliste
```

Benutzerliste: Sie umfaßt die Namen derjenigen Personen, die das entsprechende Zugriffsrecht erhalten sollen. Als Name ist der Login-Name anzugeben, den der Datenbankverwalter den einzelnen Anwendern zugewiesen hat.
Paßwortliste: Hier sind sämtliche Paßwörter anzugeben, die man in der Benutzerliste berücksichtigt hat. Die Reihenfolge der Paßwörter muß mit dem Login-Namen der Benutzerliste übereinstimmen. IDENTIFIED BY ist nur beim erstmaligen Übertragen des Rechts anzugeben.
Vergebene Rechte kann man zurückziehen. Der Widerruf der Rechte erfolgt über

```
REVOKE {CONNECT|RESOURCE|DBA}
FROM Benutzerliste
```

Standardmäßig hat in Oracle jeder Anwender das RESOURCE-Recht. Er kann also (ohne Zustimmung des Datenbankverwalters) Tabellen anlegen. Die Rechte eines Benutzers werden Privilegien genannt. Oracle kennt 80 verschiedene Privilegien, die einem Benutzer zugewiesen werden können. Jeder Benutzer weist sich gegenüber Oracle durch seinen Benutzernamen und sein Paßwort aus. dem Benutzername werden vom Datenbankadministrator die Privilegien zugeordnet. Eine Vielzahl von Privilegien können zu Rollen (Roles) zusammengefaßt werden. Eine Rolle wird mit dem Befehl CREATE ROLE ... angelegt und kann mit ALTER ROLE ... und DROP ROLE geänder bzw. gelöscht werden. Mit der Anlage einer Datenbank werden automatisch 3 voreingestellte Rollen (CONNECT, RESOURCE, DBA) generiert.
Nach der Installation von Oracle gibt es 3 Benutzer: SYS, SYSTEM, PUBLIC. SYS und SYSTEM haben DBA-Privileg. SYS hat das Recht Tabellen des DATA DICTIONARY zu verändern. Für diese Benutzer gibt es voreingestellte Paßwörter. PUBLIC ist eigentlich eine Benutzergruppe. Jeder Benutzer wird automatisch Mitglied der Gruppe PUBLIC.
Eine Oracle-Datenbank kann im Dialog- und Stapelbetrieb angesprochen werden. Jeder Anwender muß sich bei der Arbeitsaufnahme[137] beim Datenbanksystem

[137] über en oracle-7.3 unter Solaris 2.5 im Datenbanklabor der FH Regensburg

anmelden[138]. Im Dialog wird das Systen dann gestartet durch Eingabe des Kommandos **sqlplus** am Terminal. Nach Eingabe der `USER-ID`, der `ORACLE-ID` (`connect string`) und des Paßwortes können alle SQL-Kommandos direkt am Terminal eingegeben werden. Stapelbetrieb setzt voraus, daß alle SQL-Kommandos mit Hilfe eine Textaufbereitors (Editor) in einer Textdatei abgespeichert wurden. Die 1. Zeile der Datei muß `USER-ID` und Passwort im Format "`USERID/Passwort`" enthalten. SQL wird dann gestartet durch Eingabe von "`sqlplus datei_name`"[139].

2.3.2.2 Aufbau der Datenbank

Die Datenbank besteht aus Dateien und dem Data Dictionary, das die Daten in den Dateien beschreibt.

DATABASE
Eine Oracle-Datenbank besteht aus einer oder mehreren Datenbankdateien. Eine Datenbank kann mit dem Befehl `CREATE DATABASE` ... durch den Datenbank-Administrator eingerichtet werden. Die Anweisung erzeugt und initialisert:

CONTROL FILES
mit Informationen über die Datenbank. (z.B. Name der Datenbank, Name der Logdatei, Erzeugungsdatum). Die Kontrolldatei darf weder gelöscht noch verändert werden.

DATABASE FILES
enthalten alle Daten und Information über Daten. Einer Datenbank muß mindestens ein Database File zugeordnet werden. Mit der Datenbank wird automatisch der Tablespace SYSTEM und die Tablespaces TS_ROLLBACK, TS_TOOLS, TS_TEMP, TS_USERS erzeugt. Hier wird zunächst der Database File eingeordnet. Außerdem wird hier das Oracle-Data-Dictionary abgelegt. Die DD-Tabellen und Views müssen die ersten Objekte sein, die in einer Datenbank angelegt werden. Im System-Tablespace werden später auch die Dictionaries zu den Oracle-Entwicklungswerkzeugen (Oracle*Forms) abgelegt.

REDO LOG FILES
Diese Datei speichert Kopien der Blöcke, die durch Update-Operationen verändert werden. Änderungen durch den Benutzer werden zuerst in die Redo-Log-datei geschrieben und dann mit COMMIT in der Datenbank gespeichert. Je Datenbank müssen 2 LOGFILE GROUPS und je GROUP mindetsens 2 Redo-Log-Dateien angelegt sein.
Die Struktur der Datenbank kann nachträglich über das Kommando ALTER DATABASE ... verändert werden.

DATA DICTIONARY (DD)

Das DD ist eine Sammlung von Tabellen und Views mit Informationen über die in der Datenbank vorhandenen Datenbankobjekten. Eine Überblick (einschl. einer

[138] im Mehrbenutzerbetrieb zwingend vorgeschrieben
[139] <user-id>/<paßwort>@rfhs8012_ora8

kurzen Beschreibung in englischer Sprache) erhält man mit Hilfe der folgenden `select`-Anweisung:

```
SELECT * FROM DICT;
```

Das Data Dictionary einer Oracle-Datenbank besteht aus mehreren Gruppen von Tabellen und Views:

- USER_xxx: Objekte, die nur dem jeweiligen Benutzern gehören, z.B.: `select object_name from user_objects;`
- ALL_xxx: Alle Objekte, auf die ein Benutzer zugreifen kann, z.B.: `select owner, object_name from all_objects;`
- DBA_xxx: Objekte, die nur dem DBA zugänglich sind
- V_$xxx: sog. Dynamic Performance Tabellen mit Status-Informationen über die Datenbank, die kontinuierlich während der Laufzeit aktualisiert werden.

Die Gruppenbezeichnung bildet jeweils den Vorspann für den Namen der Sicht:

USER_TABLES	
USER_CATALOG	
USER_COL_COMMENTS	
USER_CONSTRAINTS	
USER_INDEXES	
USER_OBJECTS	Alle Datenbankobjekte, die zum „User“ gehören (OBJ).
USER_TAB_COLUMNS	Spaltenbezeichnungen zu Tabellen, Sichten vom „User“ (COLS).
USER_TAB_COMMENTS	Kommentare zu Tabellen, Sichten
USER_TRIGGERS	Trigger, die der User definiert hat
USER_USERS	Information über den aktuellen User
USER_VIEWS	„Views“, definiert vom aktuellen „User“

2.3.2.3 Kommunikation zwischen Benutzer und System über SQL*PLUS

Die Benutzerschnittstelle des DB-Systems ORACLE ist **SQL*PLUS**. Die Kommunikation zwischen Anwender und System erfolgt über einen einfachen Texteditor, mit dem SQL- und SQL*PLUS-Befehle eingegeben werden können. Der Editor ist stets aktiviert, wenn die Kommandozeile mit dem Prompt `>sql` erscheint.
Eingabekonventionen sind:

- Der Text der Befehle kann sich über mehrere Zeilen erstrecken. Die <RETURN>-Taste bewirkt den Sprung in eine Zeile.
- Bei Aufzählungen mehrerer Spaltennamen werden diese durch Komma getrennt. Hier genügt nicht das Leerzeichen als Trennungssymbol
- Punkte trennen Tabellennamen von Spaltennamen, falls Spalten in mehreren Tabellen vorkommen
  Im Mehrbenutzerbetrieb können auch Tabellen gleiche Namen besitzen. Sie werden dann ebenfalls durch einen vorangestellten Punkt markiert. Davor wird die Zugriffsberechtigung gestellt
- Groß- und Kleinschreibung ist bei Befehlen und Attributangaben nicht relevant.
- Attributwerte müssen jedoch stets so geschrieben werden, wie sie in den Tabellen vorkommen. Alphanumerische Werte und Datumwerte werden in einfache Hochkommata eingeschlossen. Numerische Werte werden ohne besondere Kennzeichnung eingegeben.
- Der Abschluß einer Eingabe wird mit einem Semikolon (;) markiert. Es bewirkt[140], daß der Befehl nach dem <RETURN> sofort ausgeführt wird.
- Soll der Befehl nicht sofort ausgeführt werden, dann kann man die Eingabe durch <RETURN> am Anfang einer neuen Zeile abschließen. Dann erscheint wieder der Prompt SQL>

SQL*PLUS kann auch zum Editieren von SQL-Anweisungen (z.B. zum Korrigieren von Tippfehlern) verwendet werden. Der SQL*PLus-Befehl LIST ruft die letzte Anweisung wieder auf. Eine Markierung (*) zeigt an, welche Zeile der SQL-Anweisung von der beabsichtigten Änderung betroffen sein wird. Durch Angabe einer Zeilennummer nach LIST kann auf die zu korrigierende Zeile direkt Bezug genommen werden. Die Änderung kann durch den SQL*Plus-Befehl CHANGE realisiert werden. Durch Schrägstriche getrennt, gibt man nach CHANGE zuerst den zu korrigierenden Ausdruck und dann den korrigierten Ausdruck an. Die Anweisung kann anschließend mit `SQL>RUN` zum Ablauf gebracht werden.
Auch das Löschen, Hinzufügen von Zeilen und die Angabe von zusätzlichem Text zu einer vorhandenen Zeile ist möglich.
HELP ist ein spezieller SQL*PLUS-Befehl, der SQL- bzw. SQL*PLUS-Befehle zeigt, z. B.:

```
SQL>HELP help
```

zeigt eine Liste der Hilfe-Möglichkeiten.

```
SQL>HELP insert
```

zeigt Syntax und Beschreibung des SQL-Kommandos INSERT. Auch Sprachbeschreibungsmerkmale können gezeigt werden:

```
SQL>HELP subquery
```

[140] Falls das Semikolon vergessen wird, kann man mit RUN die Ausführung veranlassen

ORACLE speichert den letzten Befehl immer in einem temporären Speicher, dem SQL-Befehlspuffer (SQL-Buffer). Sollen Kommandos dauerhaft gesichert werden, so kann das mit dem SQL*Plus-Befehl

```
SAVE Dateiname [CREATE|REPLACE|APPEND]
```

geschehen. Der gegenwärtige Inhalt des Puffers wird mit diesem Befehl ins aktuelle Verzeichnis[141] geschrieben.
Der Rücktransfer des Befehls aus der Datei in den Puffer erfolgt über

```
GET Dateiname [LIST|NOLIST]
```

Er kann danach mit RUN ausgeführt werden. Falls der SQL-Befehl unmittelbar aus einer Datei ausgeführt werden soll, dann kann das über das Kommando START[142] Dateiname erfolgen.
Ein weiters Speicherkommando ist: `SPOOL Dateiname.` Mit Hilfe dieses Befehls können Abfrage-Ergebnisse im Standard-Text-Format dauerhaft gesichert werden. Das Kommando wirkt als Schalter. Mit `SPOOL OFF` wird die Aufzeichnung der Ergebnisse beendet.
Der SQL*Plus-Befehl `HELP` gibt während der Bearbeitung Informationen und Erklärungen auf dem Bildschirm an.

2.3.2.4 SQL-Anweisungen in Oracle

Übersicht

SQL ist die umfassende Kommandosprache des Datenbanksystems Oracle. Oracle-SQL umfaßt Standard-SQL[143]. SQL-Anweisungen können eingeteilt werden:

1) Abfragen

Es handelt sich hierbei um Kommandos zur Ermittlung von Daten aus den Datenbanktabellen. Alle Abfragen beginnen mit dem Schlüsselwort **SELECT**.

2) DML-Kommandos

DML-Kommandos werden dazu benutzt, bestehende Daten auf eine der drei folgenden Arten zu verändern :

```
- Eine neue Spalte in eine Tabelle einfügen  (INSERT)
- Daten einer existierenden Spalte ändern    (UPDATE)
- Spalten löschen                            (DELETE)
```

3) DDL-Kommandos

[141] Oracle fügt bei den meisten Implementierungen automatisch die Extension SQL an.
[142] Kurzform @
[143] vgl. 1.4.3.2

DDL-Kommandos dienen zum Generieren oder Löschen von Tabellen oder "Views", z.B.:

`CREATE TABLE` ..., `DROP TABLE` ...., `CREATE VIEW` ..., `DROP VIEW` ...

4) DCL-Kommandos

Data-Control-Language-Kommandos werden dazu benutzt, Benutzern Zugriffsrechte auf Daten der Datenbank einzuräumen oder wegzunehmen. Ein Anwender kann seine Arbeit nur dann aufnehmen, wenn ihm neben der Benutzerkennung zusätzlich entweder das DBA-, CONNECT- oder das RESOURCE-Recht übertragen wurde. Damit lassen sich im einzelnen folgende lokale Rechte wahrnehmen:

- SELECT-Recht
umfaßt ausschließlich das Recht, Datenbanktabellen zu befragen
- ALTER-Recht
umfaßt ausschließlich das Recht, die Struktur einer Tabelle zu verändern. Dazu gehören: das Löschen und/oder Hinzufügen von Spalten.
- DELETE-Recht
umfaßt ausschließlich das Recht, Datensätze einer Tabelle zu löschen.
- Index-Recht
umfaßt ausschließlich das Recht, für bestimmte Tabellen Indexe zu erstellen.
- INSERT-Recht
umfaßt ausschließlich das Recht, Datensätze für bestimmte Tabellen zu übertragen
- UPDATE-Recht
umfaßt ausschließlich das Recht, Daten in bestimmten Tabellen zu verändern

Standarmäßig hat in Oracle jeder Anwender das RESOURCE-Recht. Er kann also (ohne Zustimmung des Datenbankverwalters) Tabellen anlegen.

Anwendungen[144]

1. Erzeugen von Tabellen und Eintragen der Datenwerte

Eine Tabelle wird über das SQL-Kommando `CREATE TABLE` ... erzeugt. Die Verbindung zwischen einer Tabelle und Tablespace wird über folgende Kommandos erzeugt :

`CREATE TABLE_SPACE` *`space_name`* `DATAFILE` *`dateispez`*`;`

Mit diesem Kommando wird zunächst einmal der für die Tabelle (Datei) vorgesehene Speicherraum festgelegt. Dieses Kommando ist nötig, falls bei der Tabellendefinition auf den Speicherraum Bezug genommen wird.

`CREATE TABLE` *`tabname`* `(...) SPACE` *`space_name`*`;`

Standardmäßig werden alle erzeugten Tabellen im Tablespace USERS abgelegt. Dazu ist das Kommando

`CREATE TABLE` *`tabname`*

[144] beziehen sich auf das Anwednungsbeispiel in 1.3.3

ausreichend. Die Struktur der über CREATE TABLE ... erzeugten Tabellen kann mit dem Kommando

SQL> DESCRIBE *tabname*

in Erinnerung gerufen werden.

Die Tabellen des in dieser Übung vorgesehenen Anwendungsbeispiels können auf folgende Weise erzeugt werden:

```
drop table abteilung;
create table abteilung
(abt_id varchar2(2) not null,
 bezeichnung varchar2(40));

drop table job;
create table job
(job_id varchar2(2) not null,
 titel varchar2(30),
 gehalt number(8,2));

drop table angestellte;
create table angestellte
(ang_id varchar2(3) not null,
 name varchar2(10),
 gebdatum date,
 abt_id varchar2(2),
 job_id varchar2(2));

drop table qualifikation;
create table qualifikation
(ang_id varchar2(3),
 job_id varchar2(2));
```

In CREATE TABLE .. ist nach dem Datentyp die Angabe NULL (Standardmäßiger Default-Wert) bzw. NOT NULL möglich. Damit wird festgelegt, ob eine Spalte NULL-Werte (d.h. keine Werte) enthalten darf oder nicht. Primärschlüssel sollten grundsätzlich mit der Option NOT NULL ausgestattet sein. NULL-Werte werden in allen alphanumerischen Datentypen durch Leer-Strings der Länge 0 repräsentiert. Der Wert NULL gibt an, daß der Wert einer Spalte oder eines Ausdrucks nicht verfügbar ist oder nicht zugewiesen wurde.
Jede Tabellenspalte umfaßt Werte einer bestimmten Domäne. Domänen lassen sich über Angabe des Datentyps der Spalten und der Anzahl der Zeichen definieren. Es gibt 3 Basis-Datentypen:

NUMBER(Länge, Dezimalstellen)
Felder mit numerischen Datenwerten können definiert werden mit: NUMBER, DECIMAL, FLOAT, INTEGER und SMALLINT. Am häufigsten wird NUMBER[145] benutzt. Werte für NUMBER können sein: ganze Zahlen, Dezimalzahlen, Exponentzahlen (z.B. 1E3 für 1000), negative und positive Zahlen. NUMBER ohne Längenangabe bezeichnet ein Feld mit 40 Stellen[146]. Durch Angabe einer „Länge“ verkleinert oder vergrößert man derartige numerische Felder (maximale Länge = 105). Mit „Dezimalstellen“ wird die Anzahl der Stellen nach dem Dezimalpunkt (-komma) festgelegt.

[145] zählt aber nicht zum SQL-Standard. Es besteht die Gefahr falscher Interpretationen dieses Typs auf anderen Datenbanksystemen
[146] 40 Stellen sind im Hauptspeicher besetzt, gleichgültig, ob die 40 Stellen benötigt werden oder nicht

Oracle verfügt über eine Reihe mathematischer Funktionen, z.B.:

Funktion:	**Beschreibung**:
ABS	Absolutbetrag
GREATEST(X;Y)	Maximum von X, Y
LEAST(X,Y)	Minimum von X, Y
ROUND(X,n)	Rundung von X auf n Dezimalstellen
TO_NUMBER(X)	konvertiert String X in die entsprechende Zahl
TRUNC(X,n)	schneidet die Mantisse bis auf n Dezimalen ab

Außerdem sind auf numerische Werte die relationalen Operatoren <, >, =, >=, <=, != mit der üblichen Bedeutung definiert. Arithmetische Ausdrücke können in der SELECT-, WHERE-, ORDER BY- und der HAVING-Klausel auftreten.

CHAR(Länge)
umfaßt alphanumerische Zeichen (maximale Länge: 255 Bytes). Die Angabe der Länge ist obligatorisch.
Zeichen-Konstanten werden in einfachen Anführungszeichen geschrieben. Als Operation ist die Verkettung definiert, die mit "||" beschrieben wird.
ORACLE verfügt über eine Reihe wichtiger Funktionen zur Bearbeitung lexikalischer Daten, z.B.:

Funktion:	**Beschreibung**:
LENGTH(S)	Länge des Wertes der Zeichenkette S
UPPER(S)	konvertiert den Wert von S in Großbuchstaben
LOWER(S)	konvertiert den Wert von S in Kleinbuchstaben

VARCHAR(Länge)
entspricht den Ausführungen zu dem Datentyp CHAR. Bei VARCHAR-Feldern ist die Länge variabel, die angegebene Länge bestimmt die maximale Länge in Bytes. Felder, die mit VARCHAR definiert sind, nehmen nur soviel Speicherplatz in Anspruch, wie die Feldinhalte tatsächlich lang sind.

VARCHAR2(Länge)
unterscheidet sich gegenwärtig nicht von VARCHAR. Da Änderungen bzgl. der Vergleichlogik mit CHAR-Feldern für den Typ VARCHAR geplant sind, sollte der Typ VARCHAR2 benutzt werden.

DATE
umfaßt Datumwerte in der Form DD-MON-YY[147] (Standardlänge: 7), die von 4712 v. Chr bis 4712 nach Chr. datiert werden können.Ein Datum besteht aus Tagesdatum und Tageszeit und wird in 7 Bytes gespeichert:

Byte 1	Jahrhundert	19
Byte 2	Jahr	94
Byte 3	Monat	11
Byte 4	Tag	17
Byte 5	Stunde	18
Byte 6	Minute	22
Byte 7	Sekunde	55

Im Default-Format[148] wird ein Datum folgendermaßen beschrieben:

[147] Standard-Datum-Format

DD: zweistellige Angabe der Monatstage
MON: die ersten 3 Buchstaben des (englischen) Namens geben den Monat an
YY: Die Jahreszahl besteht wieder aus 3 Stellen

Die Datumsarithmetik erlaubt folgende Operationen:

```
Datum + ganzzahlige_Anzahl_Tage = neues_Datum
Datum – ganzzahlige_Anzahl_Tage = neues Datum
Datum - Datum = Anzahl_Tage_dazwischen
```

ORACLE-SQL kennt einen speziellen Bezeichner SYSDATE, der das jeweilige Systemdatum liefert.

Neben den drei Basis-Datentypen kann eine Spalte noch in Spezialformaten definiert sein:

LONG
umfaßt alphanumerische Zeichen (maximale Länge bis zu 2GBytes) und dient zur Aufnahme eines längeren, nicht weiter formatierten Textes. Bei der Anwendung von LONG sind einige Bedingungen zu beachten:

- Es darf nur eine Spalte je Tabelle als LONG spezifiziert sein
- LONG-Spalten können nicht in WHERE-, DISTINCT- oder GROUP BY-Klauseln benutzt werden. Sie können nicht in SELECT-Befehlen mit UNION, INTERSECT oder MINUS eingesetzt werden. Auch ORDER BY, CONNECT BY funktioniert hier nicht.
- CHAR-Funktionen können nicht in diesen Spalten verwendet werden
- Eine LONG-Spalte kann nicht indiziert werden
- Eine Tabelle mit einer LONG-Spalte kann nicht in einem CLUSTER gespeichert werden.

RAW(Länge) bzw. LONG RAW
umfaßt binäre Rohdaten (maximale Länge: 255 bzw. 2 GBytes). Mit diesem Datentyp können auch Fremdformate, z.B. digitalisierte Grafiken, in ORACLE verarbeitet werden. Für die Konvertierung von Daten aus RAW-Feldern stellt ORACLE die Funktionen HEXTORAW und RAWTOHEX zur Verfügung. Der Datentyp LONG RAW dient zur Speicherung von BLOBs. BLOBs umfassen Dokumente, Grafiken, Klänge, Videos[149].

ROWID
gibt die eindeutige Identifikationsnummer (Adresse eines Datensatzes) an, mit der ORACLE jede Zeile (Tupel) in der Datenbank intern verwaltet. Vom Benutzer kann dieser Datentyp nicht vergeben werden. Er spielt bei der Tabellendefinition keine Rolle. Man kan mit einem SELECT-Befehl den Inhalt der ROWID-Felder auflisten, z.B.:

```
SELECT ROWID, Ang_ID
FROM Angestellte
WHERE Ang_ID = ‘A1’;
```

Das „desc“-Kommando von SQL*PLUS zeigt, wie die Tabellen aufgebaut wurden:

```
desc abteilung;
```

[148] Wird einer Datumsspalte eine Zeichenfolge zugewiesen, die nicht dem Default-Format entspricht, gibt Oracle wahrscheinlich eine Fehlermelfdung zurück
[149] alle Arten von Binärdateien

```
Name                           Null?    Type
 ------------------------------ -------- ----
 ABT_ID                         NOT NULL VARCHAR2(2)
 BEZEICHNUNG                             VARCHAR2(40)

desc job;

Name                           Null?    Type
 ------------------------------ -------- ----
 JOB_ID                         NOT NULL VARCHAR2(2)
 TITEL                                   VARCHAR2(30)
 GEHALT                                  NUMBER(8,2)

desc angestellte;

Name                           Null?    Type
 ------------------------------ -------- ----
 ANG_ID                         NOT NULL VARCHAR2(3)
 NAME                                    VARCHAR2(10)
 GEBDATUM                                DATE
 ABT_ID                                  VARCHAR2(2)
 JOB_ID                                  VARCHAR2(2)

desc qualifikation;

Name                           Null?    Type
 ------------------------------ -------- ----
 ANG_ID                                  VARCHAR2(3)
 JOB_ID                                  VARCHAR2(2)
```

Es folgt das Eintragen der Tabellenwerte:

```
insert into abteilung values
('KO','Konstruktion');
.....................

insert into job values
('KA','Kaufm. Angestellter',3000.00);
.....................................

insert into angestellte values
('A1','Fritz','02-JAN-50','OD','SY');
.....................................

insert into qualifikation values
('A1','SY');
............
```

etc.

2. Anfragen an die Datenbank

Selektion

```
select * from angestellte
where gebdatum like '%NOV-55';
```

Projektion

1) Jede Tabelle enthält ein Feld mit dem Namen ROWID. In ihm steht die Adresse eines Datensatzes. Die Information umfasst 3 Teile: Blocknummer der Partition, Nummer der Zeile, Nummer der "data file". Man kann mit einem normalen SELECT-Befehl den Inhalt der ROWID-Felder auflisten:

```
select rowid from angestellte;

ROWID
------------------
000000F3.0000.0002
000000F3.0001.0002
000000F3.0002.0002
000000F3.0003.0002
000000F3.0004.0002
000000F3.0005.0002
000000F3.0006.0002
000000F3.0007.0002
000000F3.0008.0002
000000F3.0009.0002
000000F3.000A.0002
000000F3.000B.0002
000000F3.000C.0002

13 rows selected.
```

2) Projektion einer virtuellen Spalte

```
select name, 'hat' "hat", ang_id
from angestellte
order by name;

NAME       hat ANG
---------- --- ---
Anton      hat A12
Emil       hat A5
Erna       hat A7
Fritz      hat A1
Gerd       hat A4
Josef      hat A13
Maria      hat A14
Rita       hat A8
Tom        hat A2
Ute        hat A9
Uwe        hat A6
Werner     hat A3
Willi      hat A10

13 rows selected.
```

Zur Bildung virtueller Spalten stehen die arithm. Operationen + - * / () und der Verkettungsoperator zur Verfügung. Falls mehrere Operationen in einem Argument stehen wird von links nach nach rechts gerechnet. Punktrechnung geht vor Strichrechnung einer Operation. Der Dateityp des Ergebnisses richtet sich

nach dem im Ausdruck enthaltenen genauesten Datentyp (`float` vor `decimal` vor `integer`).

```
select name, sysdate, gebdatum,
 to_char(sysdate,'yy') - to_char(gebdatum,'yy')
from angestellte;

select name, sysdate, gebdatum,
 to_char(sysdate,'yy') - to_char(gebdatum,'yy')
from angestellte
order by to_char(sysdate,'yy') - to_char(gebdatum,'yy');

NAME     SYSDATE   GEBDATUM  TO_CHAR(SYSDATE,'YY')-TO_CHAR(GEBDATUM,'YY')
---------- --------- --------- ----------------------------------------
Maria    26-DEC-97 17-SEP-64                                         33
Ute      26-DEC-97 08-SEP-62                                         35
Emil     26-DEC-97 02-MAR-60                                         37
Rita     26-DEC-97 02-DEC-57                                         40
Willi    26-DEC-97 07-JUL-56                                         41
Gerd     26-DEC-97 03-NOV-55                                         42
Erna     26-DEC-97 17-NOV-55                                         42
Uwe      26-DEC-97 03-APR-52                                         45
Josef    26-DEC-97 02-AUG-52                                         45
Tom      26-DEC-97 02-MAR-51                                         46
Fritz    26-DEC-97 02-JAN-50                                         47
Werner   26-DEC-97 23-JAN-48                                         49
Anton    26-DEC-97 05-JUL-48                                         49

13 rows selected.
```

Mit der Funktion `to_char(<date>,<format>)` ist die Konvertierung[150] aus dem Standardformat in andere Formatdarstellungen möglich. Mögliche Formate, in die konvertiert werden kann, sind bspw. `DD.MM.YYYY`, `MM/DD/YY`.

Verbund

Die Operation Verbund richtet sich im Datenbanksystem Oracle nach folgendem Algorithmus:

1. Das kartesische Produkt der am Verbund (JOIN) beteiligten Tabellen wird gebildet.
2. Alle nicht der Join-Bedingung entsprechenden Datensätze werden aus dem kartesischen Produkt gestrichen.
3. Alle nicht der (den) Restriktion(en) entsprechenden Datensätze werden gestrichen.

1) Bestimme die Abteilungen, in denen Angestellte mit der Berufsbezeichnung '`Systemplaner`' arbeiten und die nach `1950` geboren sind

```
select distinct abteilung.bezeichnung
from abteilung, angestellte, job
where angestellte.abt_id = abteilung.abt_id and
angestellte.job_id = job.job_id and
job.titel = 'Systemplaner' and to_char(gebdatum,'yy') > 50;
```

2) `INNER JOIN`: „Bestimme die Angestellten, die im gleichen Jahr geboren sind“.

```
select a.ang_id, a.name, b.ang_id, b.name
from angestellte a, angestellte b
where to_char(a.gebdatum,'yy') = to_char(b.gebdatum,'yy') and
a.name <> b.name;
```

[150] vgl. e) eingebaute Funktionen, Formatangaben

```
ANG NAME       ANG NAME
--- ---------- --- ----------
A12 Anton      A3  Werner
A3  Werner     A12 Anton
A13 Josef      A6  Uwe
A6  Uwe        A13 Josef
A7  Erna       A4  Gerd
A4  Gerd       A7  Erna

6 rows selected.
```

3) Verbundanweisung kombiniert mit einer Unterabfrage, die ein Ergebnis liefert: „Stelle eine Liste von Angestellten zusammen, die am meisten verdienen“.

```
select angestellte.ang_id, angestellte.name
from angestellte, job
where angestellte.job_id = job.job_id and
 job.gehalt = (select max(job.gehalt) from job);
```

4) Verbundanweisung kombiniert mit einer Unterabfrage, die ein Ergebnis liefert: „Bestimme alle Angestellten, deren Gehalt den Durchschnitt der jeweiligen Abteilung, der sie angehoeren, übertreffen“.

```
select a1.abt_id, a1.name, j1.gehalt
from angestellte a1, job j1
where a1.job_id = j1.job_id and
      j1.gehalt >
      (select avg(j2.gehalt) from angestellte a2, job j2
       where a2.job_id = j2.job_id and
             a2.abt_id = a1.abt_id)
order by a1.abt_id;
```

5) Verbundanweisung kombiniert mit einer Unterabfrage, die mehr als ein Ergebnis liefert: „Gesucht werden alle Angestellten, die ein groesseres Jahresgehalt haben als jeder Angestellte mit dem Beruf Operateur“.

```
select a.name, j.gehalt * 12 Jahreseinkommen
from angestellte a, job j
where a.job_id = j.job_id and j.gehalt * 12 > ALL
(select j.gehalt * 12
 from angestellte a, job j
 where a.job_id = j.job_id and j.titel = 'Operateur');
```

6) Verbundanweisungen mit Gruppenbildung: „Bestimme eine Tabelle in der folgenden Weise. Zunächst werden alle Mitarbeiter nach Abteilungen und dann nach Berufen aufgeführt. Zähle jeden Mitarbeiter (Anzahl) in der soeben definierten Klasse und gib fuer die Klasse das Jahresdurchschnittsgehalt an. Die Ausgabe soll in eine Tabelle mit folgenden Spaltenüberschriften erfolgen:

```
abt_id   job.titel Anzahl Jahresdurchschnittsgehalt

select abteilung.abt_id, job.titel, count(ang_id) Anzahl,
       avg(job.gehalt) * 12 Jahresdurchschnittsgehalt
from abteilung, angestellte, job
where abteilung.abt_id = angestellte.abt_id and
      angestellte.job_id = job.job_id
group by abteilung.abt_id, job.titel;
```

7) Mit Hilfe der `HAVING`-Klausel können Gruppen ausgewählt werden, die in die Ergebnistabelle aufgenommen werden sollen: „Finde das

Jahresdurchschnittsgehalt für alle Jobs, denen mehr als zwei Angestellte angehören. Die Ausgabe soll in eine Tabelle mit folgenden Spaltenüberschriften erfolgen“.

```
job.titel ANZAHL JAHRESDURCHSCHNITTSGEHALT

select job.titel, count(ang_id) ANZAHL, avg(job.gehalt *12)
       JAHRESDURCHSCHNITTSGEHALT
from job, angestellte
where angestellte.job_id = job.job_id
group by job.titel
having count (angestellte.ang_id) > 2;
```

8) Verbundanweisungen mit virtuellen Spalten: „Berechne das tägliche (Taeglich) und stündliche (STUENDLICH) Gehalt der Mitarbeiter der Abteilung ‘Organisation und Datenverarbeitung‘. Monatlich fallen 22 Arbeitstage an, die tägliche Arbeitszeit beträgt 8 Stunden.

```
select angestellte.name, gehalt MONATLICH, round (gehalt / 22,2)
       TAEGLICH, round (gehalt / (22 * 8), 2) STUENDLICH
from angestellte, job, abteilung
where abteilung.bezeichnung = 'Organisation und Datenverarbeitung'
and abteilung.abt_id = angestellte.abt_id
and angestellte.job_id = job.job_id;

NAME       MONATLICH  TAEGLICH STUENDLICH
---------- --------- --------- ----------
Werner          3000    136.36      17.05
Fritz           6000    272.73      34.09
Anton           6000    272.73      34.09
Ute             6000    272.73      34.09
```

Der **OUTER JOIN** wird in ORACLE über die WHERE-Klausel bestimmt, z.B.:

```
.......
WHERE Spaltenname_1(+) = Spaltenname_2
```

Datensätze der mit durch die Markierung „*Spaltenname_1*(+)“ in der JOIN-Bedingung gekennzeichneten Tabelle, denen kein Datensatz aus der zweiten Tabelle (, gekennzeichnet durch „*Spaltenname_2*“ in der JOIN-Bedingung,) zugeordnet werden kann, werden mit einem imaginären Datensatz, der nur aus Nullwerten besteht, verbunden.
Bsp.:

1. Finde heraus, ob gegen die referentielle Integrität verstossen wurde: „Gibt es in Qualifikation eine job_id die nicht job definiert wurde“.

```
select j.job_id, q.job_id
from job j, qualifikation q
where j.job_id(+) = q.job_id;

JO JO
-- --
IN IN
IN IN
IN IN
IN IN
KA KA
KA KA
KA KA
   OP
   OP
```

```
PR PR
PR PR
PR PR
SY SY
SY SY
SY SY
SY SY
TA TA
TA TA

18 rows selected.
```

2. Gibt es in Qualifikation eine ang_id, die nicht in angestellte definiert ist

```
select a.ang_id, q.ang_id
from angestellte a, qualifikation q
where a.ang_id(+) = q.ang_id;
```

Oracle unterstützt (ab Version 9i) auch die ANSI-SQL Syntax. Es stehen nun auch folgengende JOIN-Typen zur Verfügung:

- CROSS JOIN
- Natuaral JOIN
- Join mit USING-Klausel
- Outer-Join (Left, Right und Full)

Cross Join
Er entspricht dem kartesischen Produkt von 2 oder mehr ohne Join-Bedingung abgeleiteten Tabellen.

```
select name, abt.abt_id, bezeichnung
from angestellte CROSS JOIN abteilung abt;
```

Natural Join
Er stützt sich auf Tabellenspalten mit gleichem Namen und Datentyp. Bei dieser Verknüpfung werden automatisch alle Spalten mit gleichem Namen und Datentyp in die Join-Bedingung eingebunden.

```
select name, abt_id, bezeichnung
from angestellte NATURAL JOIN abteilung;
```

Join mit USING-Klausel
Über die USING-Klausel kann explizit eine bestimmte Spalte für die JOIN-Bedingung angegeben werden. Auch hier müssen die Spalten in beiden Tabelle den gleichen Namen und Datentyp besitzen.

```
select name, abt_id, bezeichnung
from angestellte INNER JOIN abteilung USING(abt_id);
```

Join mit ON-Klausel
Join-Prädikate können auch mit ON definiert werden. Dies ist bspw. dann erforderlich, wenn die Spalten für die Join-Bedingung in beiden Tabellen nicht denselben Namen haben.

```
select distinct titel, gehalt
from angestellte ang INNER JOIN job j
ON (ang.job_id = j.job_id
    and titel = 'Systemplaner');
```

Bisher mußte ein INNER JOIN bspw. so angegeben werden

```
select bezeichnung, count(ang_id) anz_ang
from angestellte ang, abteilung abt
where ang.abt_id = abt.abt_id
and job_id in ('SY','IN')
group by bezeichnung;
```

Mit der ON-Klausel läßt sich das so schreiben:

```
select bezeichnung, count(ang_id) anz_ang
from angestellte ang join abteilung abt
on ang.abt_id = abt.abt_id
where ang.job_id in ('SY','IN')
group by bezeichnung;
```

Mit Hilfe der ON-Klausel lässt sich auch ein Mutiple Join realisieren:

```
select abteilung.abt_id, job.titel, count(ang_id) Anzahl,
       avg(job.gehalt) * 12 Jahresdurchschnittsgehalt
from abteilung join angestellte
on (abteilung.abt_id = angestellte.abt_id)
join job on (angestellte.job_id = job.job_id)
group by abteilung.abt_id, job.titel;
```

Neben dem Left- und Right-Outer-Join ist mit Version 9i auch noch der Full-Outer-Join möglich. Der (+) Operator kann hier nicht mehr verwendet werden, wenn die ANSI-Join-Syntax angewendet wird.

Right-Outer Join
Bisher wurde ein Right-Outer Join so formuliert:

```
select a.ang_id, q.ang_id
from angestellte a, qualifikation q
where a.ang_id(+) = q.ang_id;
```

Die neue Fassung ist

```
select a.ang_id, q.ang_id
from angestellte a right outer join qualifikation q
on a.ang_id = q.ang_id;
```

Left-Outer-Join
Bsp.:

```
select a.ang_id, q.ang_id
from angestellte a left outer join qualifikation q
on a.ang_id = q.ang_id;
```

Full-Outer-Join
Ein Full Outer Join war in den alten Oracle Versionen nicht vorgesehen. Man konnte die Operation aber bspw. so sinulieren:

```
select a.ang_id, q.ang_id
from angestellte a, qualifikation q
where a.ang_id(+) = q.ang_id
union
select a.ang_id, q.ang_id
from angestellte a, qualifikation q
where a.ang_id = q.ang_id(+);
```

Ab Oracle 8i kann man den Full Outer Join so formulieren:

```
select a.ang_id, q.ang_id
from angestellte a full outer join qualifikation q
on a.ang_id = q.ang_id;
```

Unterabfragen

Sie lassen sich ab Oracle 9i auch in der SELECT- bzw. in der FROM-Klausel verwenden:

SELECT *spalte(n)*, (*einwertige_Unterabfrage*) [*spaltenAlias*]
FROM *tabelle(n)* | (*mehrwertige_Unterabfrage*) *tabellenAlias*
WHERE (e*inwertige Unterabfrage*) *op*[151] (*ein-/mehrewertige_Unterabfrage*)
GROUP BY gruppierungsfunktion
HAVING *ausdruck op* (e*in-/mehrwertige_Unterabfrage*)
ORDER BY *sortierkriterium*

Es gelten folgende Regeln für die Verwendung von Unterabfragen:

- In einer SELECT-Klausel darf die Unterabfrage nur eine Zeile mit einem einzigen Spaltenwert zurückgeben.
- In einer FROM-Kausel kann die Unterabfrage eine oder mehrere Zeilen zurückgeben.
- In einer WHERE-Klausel muß die Unterabfrage auf der linken Seite des Vergleichsoperators (op) eine einzelne Zeile und einen einzelnen Spaltenwert zurückgeben.. Die Unterabfrage auf der rechten Seite kann einen einzelnen Wert aus einer oder mehreren Zeilen zurückgebeb, abhängig vom Typ des benutzten Vergleichsoperators. In der WHERE-Klausel gibt es verschiedene Unterabfragen, die es gestatten, daß mehrere Spalten sowohl auf der linken als auch auf der rechten Seite des Vergleichsopeartors vorhanden sein können.
- In einer HAVING-Klausel kann die Unterabfrage abhängig vom verwendeten Vergleichsoperator eine Spalte aus einer oder mehreren Zeilen zurückgeben

Eine Unterabfrage selbst wird aus einer SELECT-Anweisung erzeugt, die an den genannten Stellen wiederum verschachtelte Unterabfragen enthalten kann. Zusätzlich sollten folgende Punkte beachtet werden:

- Eine Unterabfrage muß in runde Klammern eingeschlossen sein.
- An eine Unterabfrage in einer FROM-Klausel sollte ein Tabellenalias vergeben werden, um den Spaltennamen von der äußeren Abfrage unterscheiden zu können, falls es Mehrdeutigkeiten gibt.
- Eine ORDER BY-Klausel ist in einer Anwendung für eine verschachtelte Unterabfrage nicht gestattet. Die äußere Abfrage ist der einzige Bestandteil der Anweisung, die eine ORDER BY-Klausel verwendet.
- Unterabfragen können bis zu einer Tiefe von 255 Ebenen verschachtelt werden. Die Anzahl der Ebenen, die in einer Unterabfrage verwendet werden, können zu einer komplexen Anweisung und zu einer schlechten Leistung führen. SQL-Anweisungen können durch Verwendung von gespeicherte SQL- oder Java-Funktionen vereinfacht werden.

Unterabfragen lassen sich einteilen in

- verschachtelte Unterabfragen
- abhängige Unterabfragen

---

[151] Geeignete Vergleichsoperatoren sind
einwertige Vergleichsoperatoren: =, >, <, >=, <=, <>
mehrwertige Vergleichsoperatoren: IN, >ANY, <ANY, =ANY
Falls eine Unterabfrage mehrere Zeilen zurückgibt und ein einwertiger Vergleichsoperator verwendet wird, dann tritt ein Laufzeitfehler auf.

Verschachtelte Unterabfragen
Sind unabhängig von ihren übergeordneten Abfragedaten und werden einmal vor ihrer übergeordneten Abfrage ausgeführt
Abhängige Unterabfragen
Eine abhängige Unterabfrage hängt von den Informationen ihrer übergeordneten Abfrage ab. Die Daten der übergeordneten Zeile werden verwendet, um die abhängige Unterabfrage auszuwerten, die dann ein Ergebnis zurückgibt, das in der übergeordneten Abfrage verwendet werden kann. Ein abhängige Unterabfrage muß einmal für jede Zeile ausgeführt werden und wird von ihrer übergeordneten Abfrage verarbeitet.
Veranschaulichung einer abhängigen Unterabfrage in Pseudocode:

WHILE (Zeile in übergeordneten Abfrage vorhanden)
LOOP
Führe abhängige Unterabfrage mit Daten aus einer Zeile der übergeordneten Abfrage aus
Die abhängige Abfrage gibt Ergebnisse an die übergeordnete Abfrage zurück
Die übergeordnete Abfrage wendet einen Vergleichsoperator auf das Ergebnis einer untergeordneten Abfrage an
IF Ergebnis des Vergleichs wahr für die aktuelle Zeile THEN beziehe den Resultset ein
ELSE entferne die übergeordnete Zeile aus dem Resultset
END IF
Hole die nächste Zeile der übergeordneten Abfrage
END LOOP

Skalare Subqueries geben einen einzigen Wert zurück. Oracle 9i weitet ihre Verwendung auf beinahe jeden Platz aus, in dem ein Ausdruck benutzt werden kann, einschl. CASE-Ausdrücke, SELECT-Anweisung, VALUE-Klausel in einer INSERT-Anweisung, WHERE-Klausel, ORDER-Klausel, als Parameter einer Funktion.

```
select substr((select 'ABC' from dual),1,1) from dual;
```

Skalare Unterabfragen können nur eine einzige Spalte oder eine einzige Zeile zurückgeben. Gibt es mehr als eine Zeile bei der Rückgabe, dann wird ein Fehler produziert. Wird keine Zeile für die Rückgabe erzeugt, ist der Rückgabewert NULL. Der Datentyp des Rückgabewerts muß zum Datentyp mit dem er verglichen wird, passen. Skalare Unterabfragen können nicht verwendet werden für

- Default-Werte zu Spalten
- Hash-Ausdrücke für Cluster
- funktionale Index-Ausdrücke
- CHECK Constraints von Splaten
- WHEN-Bedingung von Triggern
- Group By und HAVING-Klauseln
- START WITH und CONNECT BY Klauseln

Sub-Query Factoring. In Oracle 9i ist hier die WITH-Klausel vorgesehen. Sie erlaubt einen bestimmten Teil einer Sub-Query, welcher mehrfach vorkommt einmalig zu definieren und dann mehrfach zu verwenden. Das Sub-Query Factoring gehört zum SQL-99 Standard.

```
WITH
abt_gehalt as
```

```
(select abt.abt_id, abt.bezeichnung, count(ang.ang_id),
        sum(gehalt) gehaltssumme, avg(gehalt)
 from angestellte ang inner join abteilung abt on (ang.abt_id =
abt.abt_id)
     join job j on (ang.job_id = j.job_id)
 group by abt.abt_id, abt.bezeichnung)
select * from abt_gehalt
where  gehaltssumme > (select avg(gehaltssumme) from abt_gehalt)
order by abt_id;
```

Drehen einer Tabelle (Kreuztabellenabfrage)

In einer derartigen Abfrage werden Zeilen zu Spalten und Spaten werden zu Zeilen Solche Tabellen werden Kreuztabellen genannt.
Der folgende View `angabt` zeigt Abteilungsbezeichnungen als Spaltenbezeichner und die Titel der Angestellten als Datensätze. Dazu wird die Tabelle `angestellte` gedreht.

```
create view angabt as
select angestellte.ang_id, angestellte.name,
       decode(angestellte.abt_id,'KO',
              angestellte.job_id) KO,
       decode(angestellte.abt_id,'OD',
              angestellte.job_id) OD,
       decode(angestellte.abt_id,'PA',
              angestellte.job_id) PA,
       decode(angestellte.abt_id,'RZ',
              angestellte.abt_id) RZ,
       decode(angestellte.abt_id,'VT',
              angestellte.abt_id) VT
from angestellte;
```

Die `decode`-Funktion sorgt für die korrekte Zuordnung von Berufsbezeichnungen in Zeilen und Spalten:

```
desc angabt;

Name                           Null?    Type
 ------------------------------ -------- ----
 ANG_ID                         NOT NULL VARCHAR2(3)
 NAME                                    VARCHAR2(10)
 KO                                      VARCHAR2(2)
 OD                                      VARCHAR2(2)
 PA                                      VARCHAR2(2)
 RZ                                      VARCHAR2(2)
 VT                                      VARCHAR2(2)

select * from angabt;

ANG NAME       KO OD PA RZ VT
--- ---------- -- -- -- -- --
A1  Fritz         SY
A2  Tom        IN
A3  Werner        PR
A4  Gerd                  VT
A5  Emil             PR
A6  Uwe                 RZ
A7  Erna       TA
A8  Rita       TA
A9  Ute           SY
A10 Willi      IN
A12 Anton         SY
A13 Josef      SY
```

```
A14 Maria            KA

13 rows selected.
```

Decode oder Case-Funktion[152] (Bestandteil des Intermediate Levels)

Syntax: DECODE(*Wert,Bedingung_1,Ergebnis_1,*
Bedingung_2,Ergebnis_2,
...., ...,
Bedingung_n,Ergebnis_n
[, *Defaultergebnis*])

Beschreibung: Wenn der *Wert* eine der folgenden Bedingungen entspricht, wird das entsprechende Ergebnis zurückgegeben. Wenn keiner der Vergleiche zum Ergebnis führt, wird das *Defaultergebnis* zurückgegeben. Wenn es dieses nicht gibt, ist das Ergebnis NULL.

1) Filtern von Daten mit decode

```
select count(*), count(decode(job_id,'SY',1,NULL))
from angestellte;

COUNT(*) COUNT(DECODE(JOB_ID,'SY',1,NULL))
-------- ---------------------------------
      13                                 4
```

2) Ermitteln von Prozentwerten mit decode

```
select count(decode(job_id,'SY',1,NULL)) /
 count(*) * 100 Prozent
from angestellte;

PROZENT
---------
30.769231
```

CASE Statements

Das Case Statement ist eine Erweiterung der Decode-Anweisung. In der einfachsten Fassung gibt es einen Wert zuück, wenn ein „Treffer“ gefunden wird.
Bsp.:

```
select titel, gehalt,
 (CASE
  WHEN gehalt < 3000 THEN 'niedrig'
  WHEN gehalt >= 3000 and gehalt < 6000 THEN 'mittel'
  WHEN gehalt >= 6000 THEN 'hoch'
 END) Einstufung
from job;

select titel, gehalt,
 (CASE
  WHEN job_id LIKE 'SY' AND gehalt <= 6000 THEN '10%'
  WHEN job_id LIKE 'IN' AND gehalt <= 6000 THEN '15%'
  WHEN job_id LIKE 'KA' AND gehalt <= 3000 THEN '20%'
  WHEN job_id LIKE 'TA' AND gehalt >= 3000 THEN '20%'
 END) Gehaltserhoehung
from job;
```

[152] Bestandteil des Intermediate Levels des SQL-Standards

Die Case-Funktion läßt sich anstelle von decode einsetzen. Die Schlüsselwörter when, then, else und end machen die Anweisung verständlicher:

```
select angestellte.ang_id, angestellte.name,
 (case angestellte.abt_id
   when 'KO' then angestellte.job_id
  end) as KO,
  (case angestellte.abt_id
    when 'OD' then angestellte.job_id
   end) as OD,
  (case angestellte.abt_id
   when 'PA' then angestellte.job_id
   end) as PA,
  (case angestellte.abt_id
    when 'RZ' then angestellte.job_id
   end) as RZ,
  (case angestellte.abt_id
    when 'VT' then angestellte.job_id
   end) as VT
from angestellte;
```

Operationen der relationalen Algebra: MINUS, INTERSECT, UNION

Operation der relationalen Algebra: `MINUS`: `MINUS` bestimmt, welche ausgewählten Werte der ersten Anweisung mit denen der zweiten und/oder weiterer Anweisungen nicht identisch sind

SELECT *Anweisungsfolge1*
MINUS SELECT *Anweisungdfolge2*
[MINUS SELECT *Anweisungsfolge3*] ....

```
select angestellte.ang_id from angestellte
minus
select angestellte.ang_id from angestellte
where angestellte.abt_id = 'OD';
```

Operation der relationalen Angebra: `UNION`

```
select angestellte.ang_id from angestellte
where angestellte.abt_id = 'OD'
union
select angestellte.ang_id from angestellte
where angestellte.abt_id = 'PA';
```

Operation der relationalen Algebra: `INTERSECT`: `INTERSECT` bestimmt, ob die ausgewählten Werte der ersten Anweisung mit denen der zweiten und/oder weiterer Anweisungen identisch sind. Übereinstimmungen werden am Bildschirm ausgegeben.

```
SELECT Anweisungsfolge1
       INTERSECT SELECT Anweisungsfolge2
       [INTERSECT SELECT Anweisungsfolge3] ...
```

```
select  angestellte.job_id from angestellte
where angestellte.abt_id = 'OD'
intersect
select  angestellte.job_id from angestellte
where angestellte.abt_id = 'KO';
```

Hierarchische Beziehungen

Viele in Datenbanken dargestellte Objekte stehen untereinander in einer hierarchischen Beziehung. Eine hierarchische Beziehung ist die Beziehung "Vorgesetzter" in einer Firma. Damit eine hierarchische Beziehung zwischen den Sätzen einer Tabelle besteht, muß ein Feld existieren, z.B. mit dem Namen „Vorgesetzter“ in der Tabelle „Angestellte“, das den Wert NULL hat. Alle anderen Feldwerte dieses Feldes dagegen müssen eindeutig einen Satz in der Tabelle Angestellte identifizieren. Zur Auswahl einer solchen Hierarchie ist anzugeben:

1. die Beziehung zwischen oberem und unterem Knoten der Hierarchie. Die Hierarchiebeziehung wird durch die `CONNECT-BY`-Klausel bestimmt. Damit wird die Beziehung zwischen Vorgänger und Nachfolgerknoten hergestellt, die Angabe `PRIOR` ist Verweis auf den Vorgängerknoten.
2. die Wurzel der Hierarchie (des Baumes). Sie wird durch die START WITH-Klausel definiert.

Bei Baumstrukturen ist die Einführung eines Pseudo-Felds `LEVEL` möglich (analog zu SYSDATE), das die Tiefe des jeweiligen Satzes angibt. Die Wurzel des Baums hat die Tiefe 1. Für die Anwendung der `CONNECT BY` Klausel zur Abfrage von Hierarchiebeziehungen sind Veränderungen an der Tabellenstruktur über `ALTER TABLE ...` vorzunehmen.

```
ALTER TABLE Tabellenname
 {ADD (neuer_Spaltenname neuer_Spaltentyp, ...)
  | MODIFY (alter_Spaltenname neuer_Spaltentyp [NOT NULL],
            ....)}
```

Im vorliegenden Beispiel führt das zu

```
ALTER TABLE Angestellte
 ADD (Vorgesetzter VARCHAR2(3));
```

Die Datenwerte der neuen Spalte sind über die UPDATE-Anweisung zu füllen:

```
UPDATE TabellenName SET SpaltenName = Ausdruck | NULL} [, ..
]
 [WHERE Bedingung]
```

Das führt bspw. zu

```
UPDATE Angestellte SET Vorgesetzter = ‘A16’ WHERE Ang_ID = ‘A1’;
```

Auf die so erweiterte Datenbank kann die Hierarchie mit folgenden SET-anwendungen untersucht werden:

```
SELECT Ang_ID, Name, Vorgesetzter
FROM Angestellte
CONNECT BY PRIOR Ang_ID = Vorgesetzter
START WITH Ang_ID = ‘A20’
ORDER BY Name;
```

`LEVEL` kann in einer `ORDER BY` - Klausel angewendet werden, oder an jeder Stelle, wo ein Attributname zulässig ist.

```
SELECT LEVEL, Ang_ID, Name, Vorgesetzter
FROM Angestellte
CONNECT BY PRIOR Ang_ID = Vorgesetzter
START WITH Ang_ID = ‘A20’
```

```
ORDER BY Name;
```

Eine eingerückte Liste erhält man mit der Funktion LPAD (padding from left = auffüllen von links):

```
SELECT ANG_ID, SUBSTR(LPAD(' ',2*LEVEL,' ') || NAME,1,32) NAME,
       VORGESETZTER
FROM ANGESTELLTE
CONNECT BY PRIOR ANG_ID = VORGESETZTER
START WITH ANG_ID = 'A20';
```

Wählt man mit der START WITH-Klausel nicht die Wurzel, sondern einen Unterknoten der Hierarchie, dann selektiert man Teilbäume des Gesamtbaums. Im allgemeinen Fall kann man sogar mehr als eine Wurzel haben, wenn mehrere Sätze die Bedingung der START WITH-Klausel erfüllen. Auf diese Weise kann man nicht nur Bäume, sondern auch "Wälder" selektieren.
Die hierarchischen Abfragen hat Oracle in der Version 9i überarbeitet. So galten bisher die Einschränkungen:

- eine Sortierung mit der ORDER BY-Klausel übersteuerte die gefundene Hierarchie.
- Eine JOIN oder eine VIEW, die einen JOIN beinhaltete, konnte nicht verwendet werden.
- Eine Master-Detail-Beziehung mußte demzufolge in der gleichen Tabelle abgelegt sein.
- Eine Subquery war nicht erlaubt.

Join-Beziehung. Die Verwendung eine Join in einer CONNECT BY –Klausel ist nun problemlos möglich

```
select substr(lpad('/',2*level,'-') || name,1,15) "NAME",
       abt_id, bezeichnung, level
from angestellte NATURAL JOIN abteilung
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter;
```

Erstellen einer Hilfstabelle

```
create table t_mgr as select ang_id, vorgesetzter from angestellte;
select * from t_mgr;
```

Abfrage mit zusammengesetzter Master-Detail Beziehung

```
select substr(lpad ('/', 2*level,'-') || name, 1, 15) "NAME",
a.ang_id, level
from angestellte a join t_mgr m on (a.ang_id = m.ang_id)
start with m.vorgesetzter is null
connect by prior a.ang_id = m.vorgesetzter;

drop table t_mgr;
```

Selbst eine connect-by Klausel auf einen View, der einen join beinhaltet, kann nun verwendet werden

```
create view v_ang_abt
(ang_id, name, vorgesetzter, job_id, abt_id, bezeichnung)
as
select ang_id, name, vorgesetzter, job_id, abt_id, bezeichnung
from angestellte natural join abteilung;

select substr(lpad('/',2*level,'-') || name,1,15) "NAME",
 titel, level
from v_ang_abt, job
```

```
where v_ang_abt.job_id = job.job_id
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter;

drop view v_ang_abt;
```

Resultate einer hierarchischen Anwendung konnten über ORDER BY nicht sinnvoll sortiert werden, z. B.

```
select substr(lpad('/',2*level,'-') || name,1,15) "NAME",
       abt_id, bezeichnung, level
from angestellte NATURAL JOIN abteilung
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter
order by name;
```

Ab Oracle 9i steht die ORDER SIBLINGS BY –Klausel zum Sortieren bereit. Das Schlüsselwort SIBLINGS bewirkt, dass beim Sortiervorgang die vorhanden Hierarchie berücksichtigt wird.

```
select substr(lpad('/',2*level,'-') || name,1,15) "NAME",
       abt_id, bezeichnung, level
from angestellte NATURAL JOIN abteilung
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter
order siblings by name;
```

Neu ist auch die SYS_CONNECT_BY_PATH –Klausel. Sie veranschaulicht den hierarchischen Weg zum Root-Element. Der zweite Parameter gibt das Trennzeichen zwischen den verschiedenen Stufen an.

```
select substr(sys_connect_by_path(name,'/'),1,30) "name", level
from angestellte
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter
order siblings by name;
```

Eine Sub-Query in der CONNECT BY –Klausel ist weiterhin nicht erlaubt, z.B:

```
select name, ang_id, vorgesetzter
from angestellte
start with vorgesetzter is null
connect by prior ang_id = (select vorgesetzter
 from angestellte
 where name = 'Christian');
```

Neu ist nur die Möglichkeit eine Sub-Query in der FROM-Klausel anzusprechen.

```
select substr(lpad('/',2*level,'-') || name,1,15) "NAME",
       abt.abt_id, bezeichnung, level
from angestellte ang, (select bezeichnung, abt_id from abteilung) abt
where abt.abt_id = ang.abt_id
start with vorgesetzter is null
connect by prior ang_id = vorgesetzter;
```

3. Die Ausführung von SQL-Befehlen

Der Cursor

Die Abarbeitung eines SQL-Befehls (auf einem Oracle7Server) erfolgt in mehreren Phasen, für die zum Zwischenspeichern eine spezielle Datenstruktur, der Cursor

benötigt wird. Eine Anwendung kann (bis auf Beschränkungen des virtuellen Adressraums und des Initialisierungsparameters `open_cursors`) beliebig viele Cursors eröffnen. Weiterhin werden zur Parameterübergabe bzw. zur Aufnahme des Resultats (Binde-) Variable in Anwendungen benutzt. Das Vorgehen zum Binden von Variablen unterscheidet sich nach Art des Clients (Embedded SQL, OCI, usw.).

Der PARSE-Befehl
dient zur Vorbereitung der Ausführung eines SQL-Befehls. Der Oracle-Server arbeitet nach der Methode „dynamisches SQL“, d.h.: SQL-Befehle der Anwendungen sind dem OracleServer vor der Ausführung nicht bekannt. Zur Optimierung der Performance des dynamischen SQL wird beim Server ein einmal bestimmter Ausführungsplan in dem dafür vorgesehenen Teil des SGA (Shared Pool) abgelegt. In der PARSE-Phase wird der zu bearbeitende SQL-Befehl an den Server (als Zeichenkette) geschickt und dort analysiert. Befindet sich der Befehl im Shared Pool und kann er verwendet werden, dann wird die PARSE-Phase sofort beendet. Andernfalls erfolgt eine Syntaxprüfung, die Auswertung der betroffenen Datnbankobjekte (Tabellen, Views, usw.), eine Zugriffsprüfung und die Bestimmung des Ausführungsplans durch den Optimizer (Welches Objekt wird zuerst gelesen?, Welche Indexe werden benutzt?). Der Ausführungsplan und der SQL-Befehl werden im Shared Pool abgelegt.

Die EXECUTE-Phase
Sie kann nach einer erfolgreichen PARSE-Phase durchgeführt werden. Bei allen SQL-Befehlen (außer dem `select`-Befehl) verbirgt sich hier die komplette Befehlsausführung. Auf jedem Fall wird der Inkonsistenzzeitpunkt gesetzt, so daß die Möglichkeit besteht, den internen Zeitstempel vom Lesekonsistenzzeitpunkt (System Change Number) mit dem von evtl. Änderungen an Datensätzen zu vergleichen. Beim „`select`“-Befehl sind evtl. Sortierungen und Gruppierungen vorzunehmen, die in einem temporären Bereich so zur Verfügung gestellt werden, daß in nächsten Phase die ersten Datensätze übetragen werden können. Alle anderen Befehle werden nach dem Setzen des Lesekonsistenzzeitpunkts vollständig abgearbeitet.

Die FETCH-Phase
Sie wird nur für den `select`-Befehl ausgeführt, da alle anderen Befehle mit der EXECUTE-Phase abgeschlossen sind. Bei select-Befehlen werden in der FETCH-Phase die Ergebnisdatensätze an die Anwendung übertragen. Dabei wird aus den Daten- und Indexblöcken oder aus dem in der EXECUTE-Phase vorbereitetem Temporärbereich gelesen.

4. Optimierung von SQL-Anweisungen

Für den Zugriff auf die Daten gibt es verschiedene Wege. Das System kann eine SELECT-Anweisung bspw. über einen „`Full Table Scan`“ (sequentielles Lesen aller Datensätze), einen Index-Zugriff oder einen direkten Zugriff über die Adresse eines einzelnen Satzes (ROWID) abarbeiten. Über den jeweils besten Weg entscheidet der ORACLE-Optimizer. Für jedes SQL-Kommando legt er einen Ausführungsplan fest, der eine regelbasierte bzw. statistikbasierte Abarbeitung von SQL-Anweisungen bestimmt.

5. Vergleich Oracle-SQL gegen Standard-SQL

Kein namhafter Datenbankhersteller kann auf den ANSI-Standard von SQL verzichten. Teilweise geht die Implementierung von Oracle-SQL über Full-SQL hinaus. So sind in SQL2 nur Gruppenfunktionen bestimmt, der gesamte Bereich der arithmetischen, Character- und Konvertierungsfunktionen ist dem Datenbank-Hersteller überlassen. Ebenso offen ist die Einrichtung von Indexen und Clustern. Noch nicht in die SQL2-Definition aufgenommen ist PL/SQL, Trigger, Packages, Functions und die verteilte Verarbeitung.

2.3.2.5 Datenbankprogrammierung

ORACLE bietet folgende Möglichkeiten zur Datenbankprogrammierung an:

- die Sprache **PL/SQL**
Das ist eine Sprache mit Programmiermöglichkeiten (Schleifen, bedingten Verzweigungen, explizite Fehlerbehandlung)

- **Constraints**
Integritätsregeln lassen sich zentral in Form von Constraints im Data Dictionary festlegen. Die Definition erfolgt mit der Bestimmung der Datenstrukturen über CREATE TABLE ....

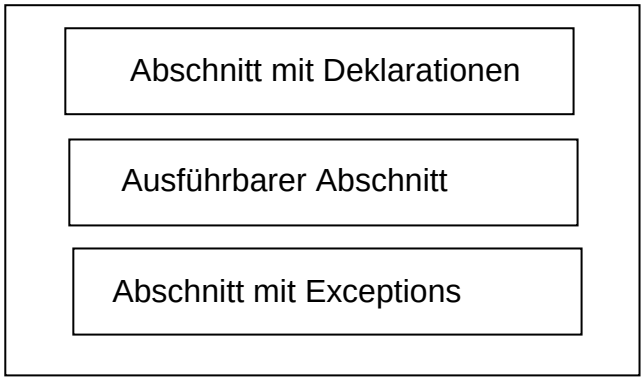
- **Datenbanktrigger**
Trigger sind einer Tabelle zugeordnete PL/SQL-Module, die bei DML-Aktionen gegen diese Tabelle ausgeführt werden. Es handelt sich um eigenständige, im Data Dictionary unkompiliert abgelegte Objekte. jede Tabelle kann bis zu 12 verschiedene Trigger haben, die sich durch unterschiedliche Auslöseereignisse (INSERT, UPDATE, DELETE), Auslösezeitpunkte (before, after) und im Typ (Befehls-, Datentyp) unterscheiden

- **Stored Procedures**, **Functions** und **Packages**
PL/SQL-Module lassen sich in kompilierter Form unter einem bestimmten Namen als Objekt in der Datenbank abspeichern.
„Stored Procedures“ sind Prozeduren, die der Entwickler mit Ein- oder Ausgabeparametern versehen kann. Eine derartige Prozedur kann über ihren Namen aufgerufen werden.
„Functions“ unterscheiden sich von „Procedures“ nur dadurch, daß sie einen Rückgabewert an die rufende Programmeinheit liefern.
„Packages“ fassen eine logisch zusammenhängende Sammlung von Prozeduren und Funktionen zusammen und legen sie in der Datenbank ab.

2.3.2.5.1 PL/SQL

a) Aufbau eines PL/SQL-Programms

PL/SQL ist eine blockorientierte Sprache. Ein PL/SQL-Programm besteht aus Prozeduren, Funktionen oder anonymen Blöcken und ist generell so aufgebaut:




```
Definitionskopf
Deklarationsteil
BEGIN
...........
 BEGIN
 ...........
 ...........
 EXCEPTION
  Ausnahmebehandlung
 END;
END;
```

Auf oberster Ebene der Struktur eines PL/SQL-Blocks können sich ein optionaler Abschnitt mit Deklarationen, ein ausführbarer Abschnitt und ein optionalen Abschnitt für die Behandlung von Exceptions und Fehlern von PL/SQL bzw. SQL befinden.

Abb.: Struktur auf oberster Ebene eines PL/SQL-Blocks

Der **Definitionskopf** bestimmt, welche Aufgaben das PL/SQL-Programm erfüllen soll. Es kann sich bspw. um einen Datenbank-Trigger, ein Paket, eine Prozedur oder eine Funktion handeln. Eine Prozedur wird mit dem reservierten Wort PROCEDURE, eine Funktion mit FUNCTION eingeleitet. Danach kann eine Parameterliste angegeben werden. Bei Funktionsdefinitionen muß zusätzlich der Datentyp des Rückgabewerts durch die RETURN-Klausel angegeben werden. Ohne Defintionskopf ist das PL/SQL-Programm ein anonymer Block. Dieser ist vergleichbar mit einer Prozedur ohne Parameter. Anonyme Blöcke kommen häufig in Skripten vor, die in einer SQL*Plus-Sitzung ausgeführt werden.
Bsp.: Einfacher, anonymer PL/SQL-Block, der einige Testdaten erzeugt

```
rem loeschen Testtabelle

drop table test_tabelle;

rem Erzeugen der Tabelle

create table test_tabelle (
 satz_nummer int,
 aktuelles_datum date);
```

```
rem PL/SQL-Prozedur zum Initialisieren der Tabelle

DECLARE
 max_satzanzahl CONSTANT int := 100;
 i                       int := 1;
BEGIN
 FOR i IN 1..max_satzanzahl LOOP
  INSERT INTO test_tabelle
              (satz_nummer, aktuelles_datum)
  VALUES
        (i, SYSDATE);
 END LOOP;
COMMIT;
END;
/
```

Deklarationsteile reservieren Speicherplatz für Variable und Konstante. Der Deklarationsteil in einem PL/SQL-Block ist optional und beginnt mit dem Schlüsselwort „`declare`“. Alle Variablen und Konstanten, die in PL/SQL-Anweisungen verwendet werden, müssen deklariert werden. Jede Deklaration einer Variablen oder Konstanten besteht aus ihrem Namen, den Datentyp und einem optionalen Initialisierungswert. Die Deklaration von Variablen und Konstanten wird (wie bei allen PL/SQL-Anweisungen) mit einem Semikolon abgeschlossen.
Der **Anweisungsteil** (Ausführungsteil) eines PL/SQL-Blocks folgt auf das Schlüsselwort „`begin`“. Jede PL/SQL-Anweisung wird durch ein Semikolon beendet. Anweisungen können sein: „Zuweisungen, Anweisungen zur Steuerung des Programmablaufs, SQL-Anweisungen, Cursor-Anweisungen.
Die Anweisungen eines PL/SQL-Programms sind in Blöcken zusammengefaßt. Jeder Block beginnt mit dem reservierten Wort `BEGIN` und endet mit dem Wort `END`. Jeder Block verfügt über einen eigenen Deklarations- und Ausnahmebehandlungsteil. PL/SQL-Blöcke können geschachtelt werden. Ein Block wird durch ein Label benannt. Das Label steht vor dem Schlüsselwort `DECLARE` des betreffenden Blocks oder vor `BEGIN`, falls kein `DECLARE` vorliegt.
Der „**Exception**“-Teil definiert einen „Exception-Handler“, der für vordefinierte und benutzerdefinierte Exceptions ausgerufen werden kann. Jeder „Exception-Handler“ besteht aus einer oder mehreren PL/SQL-Anweisungen. Eine Exception ist eine Fehlerbehandlung, die während der Ausführung eines PL/SQL-Programms auftritt. Eine Exception kann vordefiniert sein, z.B. „Auslösen einer „`dup-val_on_index`“-Exception einer doppelten Zeile in einer Tabelle mit „`insert`“. In PL/SQL sind die häufigsten Oracle-Fehler, die bei der Verwendung von SQL-Anweisungen auftreten können, als Ausnahmen vordefiniert (predefined exceptions):

Name der Ausnahme	Oracle-Fehler
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01712
LOGON_DENIED	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012
PROGRAM_ERROR	ORA-06501
STORAGE_ERROR	ORA-06500
TIMEOUT_ON_RESPONSE	ORA-00051

ZERO_DIVIDE	ORA-01476
TOO_MANY_ROWS	ORA-1422

Es können auch eigene, für die jeweilige Anwendung spezifische „Exceptions“ definiert werden. Benutzerdefinierte Ausnahmen müssen explizit über eine Anweisungsfolge ausgedrückt werden, die mit „raise *name_der_ausnahme*“ eingeleitet wird. Die Ausnahmebehandlung eines Blocks steht an dessen Ende und beginnt mit dem reservierten Wort EXCEPTION. Diesem folgen die einzelnen Ausnahmen (jeweils durch WHEN *name_der_ausnahme* THEN eingeleitet):

WHEN … -- Ausnahmezustand_1
THEN … -- Aktionen zur Fehlerbehandlung
WHEN … -- Ausnahmezustand_2
THEN … Aktionen zur fehlerbehandlung

Sobald es in PL/SQL zu einem Fehler kommt, wird eine Ausnahme signalisiert (raised). Die Abarbeitung des aktuellen Blocks wird beendet und in den Ausnahmebehandlungsteil dieses Blocks gewechselt. Jeder PL/SQL-Block kann seine eigenen Ausnahmebehandlungsteil besitzen. Nachdem die dort stehenden Anweisungen ausgeführt wurden, wird die Abarbeitung bei der nächsten Anweisung des einschließenden Blocks fortgesetzt. Es ist nicht möglich, aus dem Ausnahmebehandlungsteil in den aktuellen Block zurückzuspringen.
Für die Verarbeitung von und den Zugriff auf Fehlermeldungen gibt es die Möglichkeiten:

- Die Funktion SQLCODE ruft den zuletzt vorgefallenen SQL-Fehlerwert auf und kann ihn bspw. als Spaltenfunktion in eine Tabelle eintragen.
- Die Funktion SQLERRM ruft den zuletzt vorgefallenen ORA-Fehlerwert auf und kann ihn bspw. als Spaltenfunktion in eine Tabelle eintragen (Hinweis: Eine Oracle-Fehlermeldung ist maximal 512 Zeichen lang).

Die Definition einer eigenen Fehlermeldung (und die Ausgabe) erfolgt mit RAISE_APPLICATION_ERROR(*nummer*,*text* [,*speichern*]);
nummer: Fehlernummer im Bereich von -20000 bis -20999, Vorgabe vom Anwender
speichern: TRUE oder FALSE (Standardwert) für Speicherung des Fehlers in einer Fehlerliste

b) Deklaration von Variablen

variablen_name [CONSTANT] *datentyp* [NOT NULL][:= *ausdruck*[153]];

PL/SQL verwendet alle SQL-Datentypen. Es gibt aber eine Reihe von Ergänzungen bzw. Einschränkungen.

Numerische Datentypen
Haupttyp ist NUMBER, der mit Stellenzahl (precision) und Rundungsfaktor (scale) definiert ist. Wird die Stellenzahl ausgelassen, dann wird das Maximum angenommen. Ohne Angabe des Rundungsfaktors wird als Wert 0 verwendet. Der Wertebereich von NUMBER liegt zwischen 1.0E-120 und 9.99E125.
Aus Kompatibiltätsgründen werden folgende Subtypen zum Datentyp NUMBER unterstützt: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT.

[153] wird zur Initialisierung herangezogen

Für die Speicherung von Ganzzahlen steht der Datentyp BINARY_INTEGER mit dem Wertebereich -2147483647 bis 2147483647 zur Verfügung. Dieser Typ hat 2 Subtypen: NATURAL mit Werten von 0 bis 2147483647 und POSITIVE mit Werten von 1 bis 2147483647.

Alphanumerische Datentypen
Sie entsprechen den SQL-Datentypen, können aber in der Regel (Ausnahme: LONG) längere Zeichenketten speichern.

BOOLEAN
Variable dieses Datentyps sind TRUE oder FALSE. Zusätzlich ist auch der Wert NULL möglich[154].
Bsp.: Deklaration einer Variablen vom Typ boolean und Initialisierung dieser Variablen

```
set serveroutput on

declare
 spaeteBezahlung boolean := TRUE;
begin
 dbms_output.enable;
 if spaeteBezahlung then
  dbms_output.put_line('Die Bezahlung erfolgt spaet!');
 end if;
end;
/
```

DATE
Datumsangaben, die über den Datentyp DATE gespeichert werden, lassen sich nicht direkt in einem PL/SQL-Programm erzeugen, sondern werden immer automatisch durch Verwenden von TO_DATE in ein Datum konvertiert.

ROWID
Für die Definition von eindeutigen Schlüsseln in Tabellenspalten entspricht ROWID in PL/SQL dem gleichnamigen Felddatentyp in der Datenbank. Die Speicherung erfolgt intern binär mit fester Länge.

%type
ermöglicht den Datentyp einer Variablen als äquivalent zum Datentyp der angegebenen Spalte zu deklarieren.

variablen-name tabellen-name.spalten_name%TYPE

Bsp.: Verwenden eines benutzerdefinierten Verbunddatentyps

```
declare
 type abteilungsSatztyp is record
 ( abteilungsID  abteilung.abt_id%TYPE,
   abteilungsBEZ abteilung.bezeichnung%TYPE);
 abteilungsSatz abteilungsSatztyp;
begin
 abteilungsSatz.abteilungsID  := 'CL';
 abteilungsSatz.abteilungsBEZ := 'Controlling';
 insert into abteilung
 (abt_id, bezeichnung)
 values (abteilungsSatz.abteilungsID,
```

[154] Zustand nicht bestimmt

```
         abteilungsSatz.abteilungsBEZ);
end;
/
```

%rowtype
definiert einen zusammengesetzten Datentyp, der äquivalent zu einer Zeile der angegebenen Tabelle ist. Die zusammengesetzte Variable besteht aus dem Spalten-Namen und den Datentypen der referenzierten Tabelle.

variablen-name tabellen-name%ROWTYPE

Bsp.: Verwendung von %rowtype

```
declare
 angestelltenSatz angestellte%ROWTYPE;
begin
 dbms_output.enable;
 select *
 into angestelltenSatz
 from angestellte
 where ang_id = 'A1';
 dbms_output.put_line('Angestellten-Identifikation : '
                        || angestelltenSatz.ang_id);
 dbms_output.put_line('Angestellten-Name : '
                        || angestelltenSatz.name);
 dbms_output.put_line('Angestellten-Geburtsdatum : '
                        || angestelltenSatz.gebdatum);
end;
/
```

Komplexe Datentypen:

RECORD
entspricht einer Tabellenzeile

TABLE
Eine PL/SQL-Tabelle ist eine Sammlung von Elementen desselben Typs, die durch eine Indexnummer geordnet ist.

Gültigkeitsbereich für Variable
Ein Variable ist gültig in dem Block und allen in diesem Block geschachtelten Blöcken gültig, in dem sie deklariert wurde.

c) Sprachkonstrukte zur Belegung von Variablen und zur Steuerung des Programmablaufs

Zuweisungen, Ausdrücke und Vergleiche, Festlegen von Standardwerten für Vergleiche:

Durch den Zuweisungsoperator „**:=**“ kann einer Variablen ein Wert zugeordnet werden. Der Datentyp auf der rechten Seite des Ausdrucks muß dem Datentyp der Variablen auf der linken Seite des Ausdrucksentsprechen. PL/SQL versucht eine implizite Typkonvertierung vorzunehmen. Gelingt dies nicht, muß der Ausdruck durch eine Explizite Typkonvertierung umgewandelt sein, oder es kommt zu einem Fehler.
Ein Ausdruck besteht aus mehreren Operanden, die über Operatoren verknüpft werden. PL/SQL kennt folgende Operatoren:

Operator	Operation
**, NOT	Potenzierung, logische Verneinung
+, -	Vorzeichen
*, /	Multiplikation, Division
+, -, ||	Addition, Subtraktion, Konketenation
=, <>, !=	
<, >	kleiner als, größer als
<=, >=	kleiner als o. gleich, gößer als o. gleich
IS NULL, BETWEEN, IN, LIKE	ist Null, Wertebereich. Element in einer Menge, Mustervergleich
AND	
OR	

Operatoren gleicher Priorität werden in einem Ausdruck von links nach rechts abgearbeitet. Alle Operanden müssen den gleichen Datentyp besitzen, der mit dem Datentyp des Operators korrespondieren muß. Ausdrücke, bei denen ein Operand den Wert NULL hat, evaluieren immer zu NULL. Da jede Variable den Wert NULL annehmen kann, kennt auch jeder Boolesche Ausdruck 3 Ergebnisse: TRUE, FALSE, NULL. In PL/SQL gelten daher modifizierte Wahrheitstabellen:

AND	FALSE	TRUE	NULL
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	NULL
NULL	FALSE	NULL	NULL
OR	FALSE	TRUE	NULL
FALSE	FALSE	TRUE	NULL
TRUE	TRUE	TRUE	TRUE
NULL	NULL	TRUE	NULL
NOT	FALSE TRUE	TRUE FALSE	NULL NULL

Die einzige Möglichkeit einen Vergleich zu einem NULL-Wert durchzuführen, erfolgt über den Vergleichsoperator „`IS NULL`“. Generell werden alle Variable beim Eintritt in eine Prozedur, eine Funktion oder einen anonymen Block mit NULL initialisiert. Variable können spezifisch im Deklarationsteil von PL/SQL auf zwei Weisen initialisiert werden:

```
variablenname datentyp := initialisierungswert;
```

oder

```
variablenname datentyp DEFAULT initialisierungswert;
```

Zuweisungen zu Variablen können auch direkt über eine SELECT-Anweisung erfolgen:

```
select spalte(n) into passende_Variablenliste
from tabelle(n) where bedingung;
```

Verzweigungen:

IF-THEN-ELSE

```
IF Bedingung THEN
....  -- Anweisungen, falls die Bedingung den Wert TRUE hat
ELSE
....   -- Anweisungen, falls Bedingung den Wert FALSE oder
NULL hat
END IF;
```

Der ELSE-Zweig ist optional.
IF-THEN-ELSE-Anweisungen können beliebig geschachtelt werden.

IF-THEN-ELSIF

```
IF Bedingung_1 THEN
 ....
ELSIF Bedingung_2 THEN
 ....
ELSIF Bedingung_3 THEN
 ....
END IF;
```

Schleifen (gestalten sich strukturell genauso wie in anderen Programmiersprachen)

Unbedingte Schleifen (LOOP)
Zwei besondere Anweisungen beenden unbedingte Schleifen: EXIT und EXIT WHEN.

EXIT beendet die Schleife sofort und bewirkt: Fortsetzung des Programms mit der nächsten Anweisung nach der Schleife. Häufig ist das EXIT-Kommando in einer IF-THEN-ELSE-Anweisung eingebettet, z.B.:

```
LOOP
....  -- Anweisungen
 IF ... THEN
 ....
  EXIT;  -- sofortige Beendigung der Schleife
 END IF;
 .....  -- weitere Anweisungen
END LOOP
```

EXIT WHEN beendet Schleifen, weil eine bestimmte Bedingung eingetreten ist. Die Bedingung ist hinter dem Schlüsselwort WHEN anzugeben. Deren Wert wird bei jedem Schlüsseldurchlauf ermittelt. Es erfolgt ein Abbruch der Schleife, falls die Bedingung eingetreten ist.

```
LOOP
 .....
 EXIT WHEN Bedingung
 ....
END LOOP;
```

Bedingte Schleifen

FOR-Schleife (numerische FOR-Schleife)

Im Schleifenkopf befindet sich eine numerische Variable und ein Wertebereich. Beim 1. Durchlauf erhält die Variable die untere Grenze des Wertebereichs zugewiesen. Bei jedem Schleifendurchgang wird die Variable um 1 erhöht, bis der Variablenwert die obere Grenze überschreitet. Damit ist die Schleifenausführung beendet und mit der Ausführung der 1. Anweisung hinter der Schleife wird fortgefahren. Mit dem Schlüsselwort REVERSE kann der angegebene Wertebereich in umgekehrter Reihenfolge durchlaufen werden.

```
FOR schleifenvariable IN [ RESERVE ]untergrenze..obergrenze LOOP
.... -- Anweisungen
END LOOP
```

Bsp.: Verwendung einer for-loop-Anweisung

```
declare
 anzSchleifendurchgaenge constant positive := 100;
 i                                 positive := 1;
 j                                 positive := 1;
begin
 for i in 1..anzSchleifendurchgaenge loop
  j := j + 1;
  dbms_output.put_line('j:' || to_char(j));
 end loop;
end;
/
```

WHILE-LOOP-Schleife
Im Schleifenkopf wird eine Bedingung angegeben. Der Wert dieser Bedingung wird vor jedem Schleifendurchlauf geprüft. Ist er TRUE, dann wird die Schleife durchlaufen. Im anderen Fall erfolgt ein Abbruch der Schleife. Die Ausführung wird mit der ersten Anweisung hinter der Schleife fortgesetzt.

```
WHILE Bedingung LOOP
 ......
 ......
END LOOP;
```

d) SQL-Einbettung

Nicht erlaubt sind Anweisungen der DDL. Dagegen können an beliebiger Stelle INSERT-, UPDATE-, DELETE- und SELECT-Anweisungen[155] vorkommen. Liefert die SQL-Anweisung nur genau eine Datenzeile als Ergebnis, dann dürfen in einem PL/SQL-Programm an beliebiger Stelle INSERT-, UPDATE-, DELETE- und SELECT-Anweisungen vorkommen.
Attribute, über die Informationen zur zuletzt ausgeführten SQL-Anweisung ermittelt werden können, sind

- SQL%NOTFOUND
  ist TRUE, falls die letzte SQL-Anweisung keine Datenzeile ausgewählt, eingefügt, geändert oder gelöscht hat. Ein SELECT, das keine Datenzeile auswählt, löst die Ausnahme NO_DATA_FOUND aus. Hier kann deshalb keine Abfrage mit SQL%NOTFOUND erfolgen.
- SQL%FOUND

[155] Das Ergebnis der SELECT-Anweisung besteht nur aus genau einer Zeile. Andernfalls darf die SELECT-Anweisung nicht direkt in das Programm aufgenommen werden, sondern muß mit einem Cursor implementiert werden.

ist TRUE, falls mindestens eine Datenzeile behandelt wurde. Das Attribut evaluiert zu NULL,bevor die erste SQL-Anweisung ausgeführt wurde.
- SQL%ROWCOUNT
  liefert die Anzahl der in der letzten SQL-Anweisung verarbeiteten Datenzeilen.

„Cursor“-Technik (zur Ermittlung und Verwaltung mehrzeiliger Ergebnisse)
Ein „Cursor“ verwaltet den Zugriff auf einen Satz von Datenzeilen, der das Ergebnis einer SELECT-Anweisung ist. Cursor werden wie Variablen im Deklarationsteil eines Blocks definiert. Hier bekommt der Cursor einen Namen und eine SELECT-Anweisung zugewiesen. Nach dem Öffnen des Cursors verwaltet dieser einen Zeiger auf die aktuelle Zeile der ausgewählten Datenzeilen[156].
Der Einsatz eines Cursor umfaßt vier Arbeitsschritte:

1) Deklarieren des Cursor
```
CURSOR cursor-name
{(parameter1 parameter1-datentyp {:= default1},
 ....
parameterN parameterN-datentyp {:= defaultN})}
IS select-anweisung;
```
cursor-name: Name des Cursor
parameter1: Name des ersten an den Cursor übergebenen Parameters
parameter1-datentyp: Datentyp von parameter1
default1: optionaler Standardwert für parameter1
parameterN: Name des letzten an den Cursor übergebenen Parameters
parameterN-datentyp: Datentyp von parameterN
defaultN: optionaler Standardwert für parameterN
select-anweisung: SELECT-Anweisung, die dem deklarierten Cursor zuzuordnen ist

2) Öffnen eines Cursor
Ist der Cursor geöffnet, wird die SELECT-Anweisung ausgeführt und eine Liste mit den betroffenen Zeilen erstellt. Die Zeilen werden als aktive Menge bezeichnet.

3) Lesen von Zeilen eines Cursor
Zum Lesen der Zeilen, muß die FETCH-Anweisung ausgeführt werden, die den Wert jeder in der SELECT-Anweisung des Cursor angegebenen Spalte abruft und in einer PL/SQL-Anweisung speichert. Üblicherweise werden diese Zeilen innerhalb einer Spalte bearbeitet.
Spezielle Cursor-Attribute ermöglichen die Koordinierung der Abläufe:

- %NOTFOUND ist TRUE, falls die letzte FETCH-Anweisung keinen Datensatz mehr liefert. Der Inhalt der Variablen, die in der FETCH-Anweisung verwendet wurde, ist in diesem Fall undefiniert.
- %FOUND ist TRUE, falls bei die letzte FETCH-Anweisung eine Datenzeile gelesen wurde
- %ISOPEN ist TRUE, falls der (benannte) Cursor offen ist
- %ROWCOUNT liefert die Anzahl der bisher durch die FETCH-Anweisung gelesenen Zeilen. %ROWCOUNT erhält vor dem 1. FETCH NULL und wird bei gelesenen Zeilen um 1 erhöht.

Bsp.: Die folgenden beiden Implementierungsbeispiele zeigen, wie durch Cursor-Schleifen Sätze einer SELECT-Anweisung satzweise verarbeitet werden.

1. Implementierung:

```
set serveroutput on
rem auf Kommandozeilenebene (SQL*Plus) eingeben.
DECLARE
   string    VARCHAR2(40);
   status    INTEGER;
```

[156] vgl. Cursorkonzept

```
   cursor c1_title is
      select job.titel
         from job
      where exists
         (select * from angestellte an
          where an.job_id = job.job_id )
      order by job.titel;

BEGIN
   dbms_output.enable;
   open c1_title;
   loop
      fetch c1_title
         into string;
      if c1_title%NOTFOUND then
         exit;
      end if;

      dbms_output.put_line(string);
   end loop;
   close c1_title;
END;
/
```

2. Implementierung: Hier ist die Schleifenvariable `job_rec` implizit als „Record“ definiert, der alle im Cursor selektierten Spalten umfaßt. Die Variable bekommt je Durchlauf den jeweils nächsten Datensatz zugewiesen. Die Schleife endet, wenn kein weiterer Datensatz mehr vorhanden ist.

```
DECLARE
   string    VARCHAR2(40);
   status    INTEGER;
   cursor c1_title is
      select job.titel
         from job
      where exists
         (select * from angestellte an
          where an.job_id = job.job_id )
      order by job.titel;

BEGIN
   dbms_output.enable;
   for job_rec in c1_title loop
      string := job_rec.titel;
      dbms_output.put_line(string);
   end loop;
END;
/
```

4) Schließen eines Cursor
Ein Cursor wird geschlossen

- um ihn mit anderen Parametern wieder öffnen zu können
- um die vom Cursor belegten Ressourcen wieder freizugeben.

Falls ein PL/SQL-Programm einen Cursor nicht schließt, schließt Oracle diesen durch Beenden oder ein DISCONNECT beim Abmelden von der Datenbank.

e) Eingebaute Funktionen

PL/SQL umfaßt standardmäßig einen umfangreichen Satz von Funktionen.

Numerische Funktionen

Funktion	Rückgabewert
ABS(x[157])	Absolut-Wert von x
CEIL(x)	Kleinste Ganzzahl größer oder gleich x
COS(x)	Cosinus von x
COSH(x)	
EXP(x)	
FLOOR(x)	Größte Ganzzahl kleiner oder gleich x
LN(x)	
LOG(x)	
MOD(x,y)	Restwert der ganzzahligen Division von x durch y
POWER(x,y)	x potenziert mit y
ROUND(x,y)	x gerundet auf y Stellen
SIGN(x)	Signum von x (gibt für negative Werte -1 zurück, 1 bei 0 und positiven Werten
SIN(x)	
SINH(x)	
SQRT(x)	Quadratwurzel von x
TAN(x)	
TANH(x)	
TRUNC(x,y)	Verkürzt x auf y Stellen

Zeichenketten-Funktionen

Funktion	Rückgabewert
ASCII(s)	ASCII-Wert von s[158]
CHR(x)	Zeichen, das dem ASCII-Wert entspricht
CONCAT(s1,s2)	Verkettet die Zeichenketten s1 und s2
INITCAP(s)	Wortanfänge in Großbuchstaben
INSTR(s1,s2,x,y)	Postion des y. Auftretens von s2 ab der x. Position in s1
INSTRB(s1,s2,x,y)	wie instr, aber Suche erfolgt byteweise
LENGTH(s)	Anzahl der Zeichen in s
LENGTHB(s)	Anzahl der Bytes in s
LOWER(s)	Wandelt alle Buchstaben in Kleinbuchstaben
LPAD(s1,x,s2)	Füllt s1 am Anfang bis zur Länge x mit Zeichen s2 auf
LTRIM(s1,s2)	Entfernt am Anfang von s1 alle Zeichen, die in s2 sind.
NLS_INITCAP(s1,s2)	wie INITCAP, allerdings bzgl. der Sprache s2
NLS_LOWER(s1,s2)	wie LOWER, allerdings bzgl. der Sprache s2
NLS_UPPER(s1,s2)	wie UPPER, allerdings bzgl. der Sprache s2
NLSSORT(s1,s2)	Sortiert s2 entsprechend der Sprache s2
REPLACE(s1,s2,s3)	Ersetzt in s1 die Zeichenkette s2 durch die Zeichenkette s3
RPAD(s1,x,s2)	Füllt s1 am Ende bis zur Länge x mit Zeichen s2 auf
RTRIM(s1,s2)	Entfernt am Ende von s1 alle Zeichen, die in s2 enthalten s.
SOUNDEX(s1)	Phonetische Repräsentation von s1
SUBSTR(s,x,y)	Teilzeichenkette der Länge y ab Position von x
SUBSTRB(s1,x,y)	Teilzeichenkette der Länge y ab Position x von s1 (bytew.)
TRANSLATE(s1,s2,s3)	Ersetzt in s1 alle Zeichen, die in s2 sind durch korrespondierende Zeichen in s3
UPPER(s)	Wandelt alle Buchstaben in Großbuchstaben um

Datumsfunktionen

Funktion	Rückgabewert

[157] x, y haben den Datentyp NUMBER

[158] s, s1, s2, s3 haben den Datentyp VARCHAR2

ADD_MONTHS(d,x)	Addiert x Monate auf d[159]
LAST_DAY(d)	Datum des letzten Tages des Monats von d
MONTHS-BETWEEN(d1,d2)	Anzahl Monate zwischen d1 und d2
NEW_TIME(d,s1,s2)	Berechnet die Zeitzone s1 in Zeitzone s2 um
NEXT_DAY(d,s1)	Erster Tag der Woche nach d bzgl. Tag s1
ROUND(d,s1)	Rundet das Datum entsprechend dem Rundungsformat s1
SYSDATE	Aktuelles Datum und Uhrzeit
TRUNC(d,s1)	Schneidet die Datumsangaben von d entsprechend der Formatmaske s1 ab

Fehlerbehandlungsfunktionen

Funktion	Rückgabewert
SQLCODE	Nummer der zuletzt aufgetretenen Ausnahme
SQLERRM	Meldungstext, der zur Ausnahme laut SQLCODE gehört

SQLCODE ist ein vordefiniertes Symbol, das den Oracle-Fehlerstatus der zuvor ausgeführten PL/SQL-Anweisung enthält. Wird eine SQL-Anweisung ohne Fehler angezeigt, dann ist SQLCODE gleich 0.
SQLERRM ist ein PL/SQL-Symbol, das die mit SQLCODE verbundene Fehlermeldung enthält. Wird eine SQL-Anweisung erfolgreich durchgeführt, ist SQLCODE gleich 0 und SQLERRM enthält die Zeichenfolge `Oracle-0000:normal, successfull completition`.

Umwandlungsfunktionen

Funktion	Rückgabewert
CHARTOROWID(s)	Wandelt s in eine ROWID um
CONVERT(s1,s2,s3)	Wandelt s1 von Zeichensatz s2 nach Zeichensatz s3 um
HEXTORAW(s)	Wandelt die Kette von Hexadezimalzahlen s nach Raw um
ROWIDTOCHAR(i[160])	Wandelt i in eine Zeichenkette um
TO_CHAR(d_wert,d_format)	d_wert ist eine Datumsliteral, ein Datumswert aus einer Spalte oder ein von einer integrierten Funktion zurückgegebener Datumswert. d_format ist ein gültiges Datumsformat von Oracle
TO_CHAR(zahl[,format])	zahl ist ein numerischer Ausdruck, der umgewandelt werden soll. format ist optionales Format, das von to_char verwendet werden soll.
TO_DATE(s_wert, d_format)	s_wert ist eine Zeichenfolgenliteral, eine Zeichenfolge einer Spalte oder eine von einer zeichenfolge zurückgegebene Zeichenfolge d_format ist ein gültiges Oracle-Datumsformat
TO_MULTI_BYTE(s)	Wandelt alle Einzelbyte-Zeichen in ihr Multibyte-Zeichenäquivalent um
TO_NUMBER(s1,s2,s3)	Wandelt s1 lt. Format s2 und Sprache s3 in einen numerischen Wert um
TO_SINGLE_BYTE(s)	Wandelt alle Multibyte-Zeichen in ihr Einzelbate-Zeichen-Äquivalent um

Wird ein Datumswert umgewandelt, dann können folgende Formatangaben verwendet werden:

[159] d, d1, d2 haben den Datentyp DATE
[160] i hat den Typ ROWID

Formatangabe	Bedeutung
CC, SCC	Jahrhundert (S bedeutet: Angaben vor Christi Geburt werden mit einem Minuszeichen gekennzeichnet
YY,YYYY	die beiden letzten des Jahres bzw. vollst. Jahresangabe (Jahr und Jahrhundert). Falls einige der Ys bei der vollständigen Angabe weggelassen werden, werden die am weitesten links stehenden Ziffern des Jahres entfernt
YEAR	ausgeschriebenes Jahr z.B. NINETEEN NINETY-SEVEN
MM	Monat (01-12)
MONTH	Ausgeschriebener Name des Monats
MON	dreibuchstabige Abkürzung des Monats (JAN – DEC)
Q	Quartal des Jahres (1 – 4)
W	Woche des Monats (1 – 5)
DDD	Tag des Jahres (1-366)
DD	Tag des Monats (1-31)
D	Tag der Woche (1-7, Sonntag = 1)
DAY	ausgeschriebener Wochentag (SUNDAY – SATURDAY)
AM, PM	Meridian-Kennzeichnung
A.M, P.M	Meridian-Kennzeichnung mit Punkten
HH, HH12	Stunden des Tages (1-12)
HH24	Stunden des Tages (24 Stunden Angabe)
MI	Minute (0 - 59)
SS	Sekunden (0 – 59)
SSSSS	Sekunden seit Mitternacht (0 – 86399)

Zur Errinnerung: Eine DATE-Spalte in Oracle umfaßt das Jahrhundert, den Tag sowei Monat, Jahr, Stunde, Minute und Sekunde.
Bsp.:

Sonstige Funktionen

Funktion	Rückgabewert
DECODE(u,u1,u2[161])[162]	Vergleicht einen Ausdruck u mit den vorgegebenen Ergebnissen (u1 etc.) und gibt den entsprechenden Wert (u2 etc.) zurück
DUMP(u1,u2)	Interne Darstellung von u1 lt. Format u2
GREATEST(u,u1,u2...)	Größter Wert der Liste
LEAST(u,u1,u2,....)	Kleinster Wert der Liste
NVL(u1,u2)	Rückgabe von u2, wenn u1 Null ist, sonst u1
UID	Benutzerkennung des ORACLE-Benutzers
USER	Benutzername des ORACLE-Benutzers
USERENV(s1)	
VSIZE(u)	Anzahl Bytes der internen Repräsentation von u

f) PL/SQL-Utilities

Es handelt sich dabei um die Pakete

dbms_transaction
Alle Prozeduren und Funktionen des dbms_transaction-Pakets sind gleichzusetzen mit in SQL*PLUS angegebenen Anweisungen, die mit SET TRANSACTION beginnen
dbms_session

[161] unspezifiziert
[162] DECODE darf nur in SQL-Anweisungen verwendet werden

Die meisten hierin zusammengefaßten Routinen sind mit der ALTER SESSION-Anweisung identisch

dbms_ddl
zur Übersetzung gespeicherter Prozeduren, Funktionen, Pakete bzw. Einflußnahmen auf Optimierungsstragien bzw. zum Ünerprüfen von Namen und deren Typinformationen

dbms_lock
zur prozeduralen Steuerung von Sperren

dbms_output
zur Ausgabe von Meldungen

dbms_snap
zur Kontrolle und Steuerung von Logs bei der automatischen Verteilung von Daten über Datenbank- und Rechnergrenzen hinweg.

g) Dynamisches SQL in PL/SQL-Programmen

Das „Dynamic Package **dbms_sql**“ enthält alle notwendigen Funktionen und Prozeduren zur Verwendung des dynamischen SQL in PL/SQL-Programmen.

2.3.2.5.2 Constraints

Constraints werden in der Datendefinition angelegt. Es gibt feld- und tabellenbezogene **Constraints**. Feldbezogene **Constraints** werden direkt mit der Feldposition angelegt, tabellenbezogene **Constraints** stehen am Ende der Tabellendefinition. **Constraints** können auch einen Namen erhalten. Unbenannte Constraints erhalten von Oracle automatisch einen Namen. Unter diesem Namen findet man die Constraint dann auch im Data Dictionary.
Constraints werden bei jeder Insert-, Update- oder Delete-Aktion für die ganze Tabelle überprüft. Ein Verstoß gegen eine Constraint führt zu einem Transaktionsabbruch mit Fehlermeldung.

Spezifikation eines einfachen Constraint.

```
[CONSTRAINT name] PRIMARY KEY | UNIQUE | NOT NULL
```

Ein Constraint kann einen Namen besitzen. Falls ein solcher Name nicht angegeben wird, generiert Oracle automatisch einen Namen (Muster: SYS_C*number*)

Semantische Integrität
Zur Sicherung der semantischen Integrität können 4 Arten von Constraints eingesetzt werden:

- NOT NULL
- DEFAULT
  Die DEFAULT-Klausel enthält einen Ausdruck. Der Wert dieses Ausdrucks wird dem Attribut zugewiesen, falls explizit kein anderer Wert bereitgestellt wird. In Wertzuweisungsausdrücken der DEFAULT-Klausel dürfen SYSDATE, USER, UID oder USERENV enthalten sein. DEFAULT kann auch in den CREATE TABLE und ALTER TABLE MODIFY-Befehlen verwendet werden.
- CHECK

Hiermit können Spaltenwerte dahingehend überprüft werden, ob sie den in der CHECK-Klausel definierten Regeln entspricht. Die CHECK-Klausel kann wie eine WHERE-Klausel aufgebaut und verstanden werden. Pseudo-Spalten, Referenzen auf andere Tabellen sind allerdings nicht erlaubt.

```
[CONSTRAINT name] CHECK (bedingung)
```

- UNIQUE

Mit dieser Klausel kann für eine oder mehrere Spalten festgelegt werden, daß Datensätze einer Tabelle für diese Spalte(n) eindeutige Werte besitzen müssen. Für UNIQUE definierte Spalten wird automatisch ein UNIQUE INDEX angelegt. Deshalb können in der UNIQUE-Constraint- auch die Bedingungen für die Generierung des Index wie im CREATE INDEX Befehl enthalten sein

Entitätsintegrität

Mit der PRIMARY-KEY-Constraint kann der Primärschlüssel einer Tabelle definiert werden. Darüber werden für die Spalten der Primärschlüssel implizit die Constraints NOT NULL und UNIQUE festgelegt.

Referentielle Integrität

Sie werden über die FOREIGN-KEY-REFERENCES-Constraint gesichert. Wird eine FOREIGN-KEY-Constraint angelegt, kann ein Parent-Key nur dann gelöscht oder geändert werden, falls keine zugehörigen Datesätze in der Child-Tabelle vorhanden sind. Es dürfen nur Werte in Fremdschlüsselattributen vorkommen, für die es einen korrespondierenden Wert in der Parent-Tabelle gibt.

```
[CONSTRAINT name][FOREIGN KEY (spalte(n)]
 REFERENCES tabelle [(spalte(n))]
[ON DELETE CASCADE]
```

Wird in der FOREIGN-KEY-Constraint die Klausel CASCADE ON DELETE angegeben, so werden beim Löschen eines Parent-Satzes alle korrespondierenden Zeilen der Child-Tabelle gelöscht. Das Kaskadieren kann über mehrere Tabellen gehen. Ein kaskadierender Update wird über die Constraint-Definitionen nicht unterstützt. Gleiches gilt für Fälle, in denen bei Änderung oder Löschung eines Parent-Datensatzes die Fremdschlüsselattribute der korrespondierenden Cild-Datensätze auf `NULL` oder einen Default-Wert gesetzt werden sollen.

Bsp.: Zur Sicherung der semantischen und Entitäts-Integrität ist für die Personal-Datenbank folgendes Schema vorgesehen:

```
create table abteil
(
 abt_id varchar2(2) not null
 constraint PKab_id
 primary key,
 bezeichnung varchar2(40));

create table beruf
(
 job_id varchar2(2) not null
 constraint PBjob_id
 primary key,
 titel varchar2(30),
 gehalt number(8,2));

create table angest
(
 ang_id varchar2(3) not null
```

```
 constraint PKan_id
 primary key,
 name varchar2(10),
 gebjahr date,
 abt_id varchar2(2),
 constraint FKabt_id
 foreign key(abt_id)
 references abteil(abt_id),
 job_id varchar2(2),
 constraint FKjob_id1
 foreign key(job_id)
 references beruf(job_id));

create table quali
(ang_id varchar2(3),
 constraint FKang_id
 foreign key(ang_id)
 references angest(ang_id),
 job_id varchar2(2),
 constraint FKjob_id2
 foreign key(job_id)
 references beruf(job_id),
 constraint PKquali
 primary key(ang_id,job_id));
```

Verwaltung von Constraints
Mit dem ALTER TABLE Befehl können Constraints hinzugefügt (ADD), gelöscht (DROP) oder aktiviert (ENABLE) bzw. deaktiviert (DISABLE) werden.

```
alter table tabelle
 add (spalte datentyp [default_wert] [spalten_constraint];
alter table tabelle add (tabellen_constraint);
add table tabelle
 modify (spalte [datentyp][default_wert][spalten_constraint]
drop table tabelle [cascade_constraints]
```

2.3.2.5.3 Trigger

Oracle unterscheidet drei Trigger-Typen: DML-Trigger, Instead-of-Trigger, System-Trigger.
DML-Trigger. Anweisungen aus dem DML-Bereich von SQL wie INSERT, UPDATE, DELETE Können einen DML-Trigger auslösen. Dabei kann man bei der Programmierung des Triggers bestimmen, ob die Anweisungen vor oder nach dem auslösenden Ereignis (vor oder nach dem Eintragen, Löschen, Aktualisieren also) ausgelöst werden sollen.
Instead-of-Trigger. Auch Instead-of-Trigger reagieren auf DML-Operationen allerdings ausschließlich nur auf Sichten (relationale oder Objekt-Sichten).
System-Trigger. Der System-Trigger ist ein Werkzeug zum Abfangen von System-Ereignissen und DDL-Befehle. Systemereignisse können bspw. das Starten und Hochfhren der Datenbank, das An- und Abmelden von Benutzern sowie Fehler sein.

1. DML-Trigger

Sie sind eine spezielle Form von PL/SQL. Trigger dienen ebenso wie Constraints der Integritätssicherung. Constraints sind allerdings viel einfacher zu

programmieren. Es gibt aber einige Fälle, in denen sich Trigger-Programmierung nicht vermeiden läßt:

- Komplexe Regeln müssen über Trigger realisiert werden
- Constraints werden bei vielen INSERT-, UPDATE-, DELETE-Anweisungen überprüft. Trigger können bestimmten Ereignissen und Auswertungszeitpunkten zugeordnet werden.

Erstellen eines Trigger

```
CREATE [ OR REPLACE ] TRIGGER trigger-name { BEFORE | AFTER }
 triggering-event ON tabellen-name
[FOR EACH ROW ]
[ WHEN (bedingung) ]
      PL/SQL-Block
```

trigger-name: Name des zu erstellenden Trigger
triggering-event: INSERT, UPDATE, DELETE
tabellen-name: *Name der dem Trigger zugeordneten Tabelle*
FOR EACH ROW ist eine optionale Klausel
Bedingung: optonale Bedingung die, falls TRUE, den Trigger auslöst
PL/SQL-Block*: wird ausgeführt, wenn der Trigger ausgelöst wird (Trigger-Rumpf).*

Trigger bestehen aus folgenden Elementen:

- Trigger-Name: `CREATE | REPLACE TRIGGER Trigger_Name`
- Trigger-Zeitpunkt: BEFORE / AFTER
  Ein befehlsorientierter BEFORE- (UPDATE-) Trigger feuert genau einmal vor der Ausführung des Befehls. Analog dazu feuert ein befehlsorientierter AFTER-Trigger genau einmal nach der Ausführung des Befehls.
- Trigger-Ereignis:
```
INSERT | UPDATE [OF Spalte1, Spalte2, ... ]
       | DELETE ON Tabellen_Name
```
  Ein Trigger kann für mindestens ein oder mehrere Ereignisse definiert werden. Ist der Trigger für mehrere Ereignisse definiert, kann innerhalb des Trigger-Runpfs das jeweilige Ereignis mit `if inserting then`, `if updating then` oder `if deleting then` abgefragt werden.
- Trigger-Typ: [FOR EACH ROW]
  Ohne diese Klausel ist der Trigger befehlsorientiert, anderfalls zeilenorientiert. Ein befehlsorientierter Trigger wird genau einmal nach der Ausführung desjenigen Befehls ausgeführt, der das Ereignis auslöst. Ein zeilenorientierter Trigger wird einmal je Datensatz ausgeführt, der durch den Befehl verändert wurde. Bei zeilenorientierten Triggern ist der Zugriff auf Werte vor und nach der Änderung mit `:OLD.<Spaltenname>` und `:NEW.<Spaltenname>` möglich. Bei Update-Triggern können `OLD.<Spaltenname>` und `:NEW.<Spaltenname>`,bei INSERT-Triggern nur `:NEW.<Spaltenname>` und bei Delete-Triggern nur `:OLD.<Spaltenname>` verwendet werden.
- Trigger-Restriktion: WHEN-Prädikat
  Trigger-Restriktionen stellen Bedingungen dar, unter denen der Trigger ausgeführt werden soll oder nicht. Sie werden in Prädikaten formuliert.
- Trigger- Rumpf: Das ist ein anonymer PL/SQL-Block

Kombiniert man die 3 Ereignisse (INSERT, UPDATE, DELETE) mit den 2 erlaubten Typen, dann können je Tabelle 12 Trigger definiert werden:

- 6 Trigger auf Zeilenebene für BEFORE DELETE, BEFORE INSERT, BEFORE UPDATE, AFTER DELETE, AFTER INSERT, AFTER UPDATE
- 6 Trigger auf Anweisungsebene für BEFORE DELETE, BEFORE INSERT, BEFORE UPDATE, AFTER DELETE, AFTER INSERT, AFTER UPDATE

Bsp.: Der folgende befehlsorientierte BEFORE-Trigger speichert in einer Protokolldatei den Benutzernamen, die verwendete Anweisung und das Datum.

```
rem TABELLE EVTL. LOESCHEN
drop table protokoll;

rem TABELLE ERZEUGEN
create table protokoll
(  Benutzer     varchar2(30),
   Anweisung    varchar2(20),
   Datum        date
);

rem DATENBANK-TRIGGER ERZEUGEN
create or replace trigger audit_angestellte
before delete or insert or update on angestellte
declare
   statement    varchar2(20);
begin
   if deleting then
      statement := 'DELETE';
   end if;
   if inserting then
      statement := 'INSERT';
   end if;
   if updating then
      statement := 'UPDATE';
   end if;

   insert into protokoll
   values (USER, statement, SYSDATE);
end;
/
```

In einem Datenbank-Trigger können keine COMMIT- oder ROLLBACK-Anweisungen ausgeführt werden. Außerdem kann ein Trigger keine gespeicherte Funktion, Prozedur oder kein vordefiniertes Unterprogramm aufrufen, die eine COMMIT- oder ROLLBACK-Anweisung ausführen.

Trigger-Typen.

Ein Trigger-Typ definiert sich über die Trigger-Transaktion und die Ebene, auf der ein Trigger ausgeführt wird. Man unterscheidet: Trigger auf Zeilenebene, Trigger auf Anweisungsebene, BEFORE- und AFTER-Trigger, INSTEAD OF-Trigger, Schema-Trigger, Trigger auf Datenbankebene.
Trigger auf Zeilenebene. Sie werden für jede Zeile ausgeführt, die von einer DML-Anweisung betroffen ist. Trigger auf Zeilenebene kommen am häufigsten vor und werden in erster Linie für Anwendungen im Bereich des Daten-Auditings eingesetzt. Zu dem lassen sich mit diesem Trigger-Typ veteilte Daten synchron halten. Trigger auf Zeilenebene werden mit der for each row –Klausel im create trigger –Befehl angelegt.
Trigger auf Anweisungsebene. Sie werden bei jeder DML-Anweisung einmal ausgeführt. Trigger auf Anweisungsebene sind die Standard-Trigger und werden mit dem Befehl create trigger angelegt.
BEFORE- und AFTER –Triger. Da Trigger aufgrund von Events ausgeführt werden, kann man sie so einrichten, dass sie vor oder nach diesen Events (z.B. vor und nach inserts, updates, deletes) ausgeführt werden. Innerhalb eines Triggers können alte und neue Werte referenziert werden, die in DML-Anweisungen eingebunden sind. Der benötigte Zugriff auf die alten und neuen Daten kann den Trigger-Typ festlegen. "Alt" referenziert Daten, die vor der DML-Anweisung existieren: Aktualisierungen und Löschungen referenzieren

üblicherweise alte Werte. "Neue" Werte sind Datenwerte, die von der DML-Anweisung angelegt werden. Soll in einer eingefügten Zeile ein Spaltenwert über einen Trigger gesetzt werden, benötigt man für den Zugriff auf die neuen Daten eine BEFORE INSERT –Trigger. Ein AFTER INSERT –Trigger erlaubt kein Setzen der neu eingefügten Werte, da die Zeile bereits in der Tabelle hinterlegt wurde. AFTER-Trigger auf Zeilenebene kommen häufig bei Audit-Anwendungen zum Einsatz, da sie erst nach dem Ändern von Daten aktiv werden. Werden die Zeilen erfolgreich modifiziert, impliziert das auch die Prüfung der referenziellen Integritäts-Constraints, die für diese Tabelle definiert sind.
INSTEAD OF –Trigger. Hier wird mitgeteilt, was an Stelle der Aktion zu tun ist, die den Trigger aktiviert haben
Schema-Trigger. Für Operationen auf Schema-Ebene (create table, alter table, drop table, audit, rename, truncate, revoke) können Trigger angelegt werden. Üblicherweise bietet der Trigger-Typ an: das Verhindern von DDL-Operationen und zusätzliche Sicherheitsprüfungen beim Auftreten von DDL-Operationen.
Trigger auf Datenbankebene. Es können Trigger angelegt werden, die bei Datenbank-Events aktiviert werden. Dazu gehören: Fehler, An- und Abmelden, Starten und Herunterfahren der Datenbank.

2. Instead-of-Trigger

Dieser Trigger dient zum Überwachen von Sichten bei INSERT-, DELETE- und UPDATE-Anweisungen. Über Insted-of-Trigger wird mitgeteilt, wie die darunterliegenden Tabellen, die Bestandteil einer View sind, zu aktualisieren sind. Instead-of-Trigger lassen sich nur auf Sichten ausführen, die änderbar sind[163]. Die Sichten dürfen keine Mengenoperationen (UNION, MINUS), Aggregatfunktionen, Gruppierungen oder Verkettungen (GROUP BY oder START WITH) sowie DISTINCT enthalten.

Definition:
```
CREATE [OR REPLACE] TRIGGER trigger-name
        INTEAD OF ereignis
        [referenzklausel]
        [WHEN trigger-bedingung]
        [FOR EACH ROW]
        anweisungsabschnitt
```

2.3.2.5.4 Stored Procedures, Functions, Packages

PROCEDURES und FUNCTIONS

Eine PROCEDURE kann folgendermaßen definiert werden:

```
CREATE [OR REPLACE] PROCEDURE proc_name
[(<parameter_liste>)
IS <pl/sql_body>
/
```

- `[OR REPLACE]` : existierende Prozedurdefinition wird überschrieben
- `(<parameter_liste>)` : Deklaration der formalen Parameter

```
   (<variable> [IN | OUT |IN OUT] <datatype>,
    …..
    <variable> [IN | OUT |IN OUT] <datatype>)
```

[163] vgl. Ueb7.doc

- IN, OUT, IN OUT geben an, wie die Prozedur / Funktion auf die Parameter zugreifen kann (Lesen, Schreiben, beides)
- DEFAULT ist IN
- Bei OUT und IN OUT muß beim Aufruf eine Variable angegeben sein, bei IN ist auch eine Konstante erlaubt.
- `<datatype>` : Alle von PL/SQL unterstützten Datentypen; ohne Längenangabe (VARCHAR2 anstelle von bspw. VARCHAR2(20)
- `pl/sql_body` enthält die Definition der Prozedur in PL/SQL

Analog kann eine Funktion definiert werden:

```
CREATE [OR REPLACE] FUNCTION funct_name
[(<parameter_liste>)
RETURN <ausdruck>
IS <pl/sql_body>
/
```

- PL/SQL-Funktionen werden mit RETURN <ausdruck> verlassen, wobei <ausdruck> ein Ausdruck über PL/SQL-Variablen und –Funktionen ist. Jede Funktion muß mindestens ein RETURN-Statement enthalten

Gelöscht werden können Prozeduren bzw. Funktionen über:

```
DROP PROCEDURE prozedurname
DROP FUNCTION funktionsname
```

Im Gegensatz zu anonymen Blöcken braucht keine DECLARE-Klausel angegeben werden.

Aufrufe/Verwendung von Prozeduren / Funktionen

- Aufruf von Prozeduren im Pl/SQL-Skript: *prozedurname*(*arg1*, *arg2* , ... , *argn*)
- Aufruf von Prozeduren in SQL*PLUS: execute *prozedurname*(*arg1*, *arg2*, ... , *argn*)
- Verwendung der Funktion in PL/SQL ... *funktionsname*(*arg1*, *arg2* , ... , *argn*) wie in anderen Programmiersprachen

Im Falle von `"... created with compilation errors"` Fehler über SHOW ERRORS ausgeben lassen
Die systemeigene Tabelle DUAL kann zur Ausgabe des Ergebnisses freier Funktionen verwendet werden.

PACKAGES

Logische zusammengehörige Blöcke, Prozeduren, Funktionen können zusammengefaßt werden. Die Zusammenfassung sieht ein Interface vor, das dem Benutzer die Funktionalität bereitstellt. Diese Art Modularisierung unterstützt PL/SQL über PACKAGES. Ein PACKAGE besteht aus der Spezifikation und dem PACKAGE-Körper.

2.3.2.6 Datenbank-Verwaltung

National Language Support

Darüber werden sprachspezifische Konventionen eines Landes unterstützt. Die Implementierung erlaubt mehrsprachiges Arbeiten mit der Datenbank und den Tools.

Zugriffsrechte

ORACLE ist ein Mehrbenutzer-System. Auch andere Benutzer, nicht nur der Eigentümer, der den CREATE-Befehl für eine Tabelle gegeben hat, haben Zugriffsrechte. In ORACLE gilt zunächst das Prinzip, alles zu verbieten, was nicht ausdrücklich erlaubt ist. Zur Vergabe von Rechten muß man entweder das DBA-Recht besitzen oder Inhaber einer Tabelle[164] sein. Zur Vergabe lokaler Rechte steht der GRANT-Befehl zur Verfügung:

```
GRANT {Rechte|ALL}
ON Tabellenname
TO {PUBLIC|Benutzerliste}
[WITH GRANT OPTION]
```

Rechte: Das sind hier die lokalen Rechte. Zusätzlich kann man dem Recht UPDATE eine Spaltenliste hinzufügen. Diese Liste kann die Namen einer oder mehrere Spalten umfassen[165] und ist mit "()" einzugrenzen.
ALL: Vereinbarung, daß die unter Benutzerliste angegebenen Anwender alle lokalen Rechte erhalten
PUBLIC: Altenative Vereinbarung zu Benutzerliste. Das Recht bzw. die Rechte werden auf alle Anwender übertragen.
WITH GRANT OPTION: Vereinbarung, daß der Empfänger die erhaltenen Rechte an andere Anwender weitergeben darf.

Ein erteiltes Zugriffsrecht kann mit

```
REVOKE {Rechte|ALL}
ON Tabellenname
FROM {PUBLIC|Benutzerliste}
```

zurückgenommen werden.

Indexe und Cluster

a) Indexe
Ein Index besteht aus einer Tabelle, die zu den Werten einer Spalte (Indexspalte) die interne Adresse der zugehörigen Tabelle enthält:

- Jede Spalte (jedes Atrribut) kann indiziert werden, unabhängig von Typ und Länge (Ausnahme: Spalten vom Typ: LONG)

[164] Derjenige ist Inhaber einer Tabelle, der sie mit dem CREATE TABLE-Befehl angelegt hat
[165] beschränkt den Zugriff auf die Inhalte bestimmter Spalten

- Ein Index darf mehrere Spalten umfassen. Maximal können das 16 Spalten sein. Die Gesamtlänge darf jedoch 240 Zeichen nicht überschreiten
- Je Tabelle sind beliebig viele Indexe erlaubt.

Indexe kann der Eigentümer der Tabelle erstellen und die Benutzer, denen mit GRANT INDEX die Befugnis dazu erteilt wurde.

```
CREATE [UNIQUE] INDEX Indexname
ON     Tabelle (Attributname [,Attributname ...])
       [ Optionen ]
```

Wurde ein bestimmtes Attribut indiziert, dann erfolgt die weitere Indexverwaltung (komprimierte und unkomprimierte B-Bäume) automatisch. Die Form der SELECT-, INSERT-, UPDATE- oder DELETE-Befehle ändert sich nicht.
Mit Indexen kann man außerdem sicherstellen, daß ein bestimmter Attributwert nur einmal in einer Tabelle existiert.
Ein Index kann auch wieder gelöscht werden über: `DROP INDEX Indexname`

b) Cluster
Mit einem Cluster kann die physikalische Speicherung der Tabellen beeinflußt werden, die Attribute vom gleichen Typ, gleicher Länge und gleicher Bedeutung besitzen. Durch die Bildung eines "Cluster" wird erreicht:

- die Datensätze der beteiligten Tabellen werden auf gleiche Plattensektoren gespeichert
- jeder unterschiedliche Attributwert der betreffenden Attribute wird nur einmal gespeichert

Transaktionen

Transaktionen überführen Datenbanken von einem konsistenten Zustand in einen neuen konsistenten Zustand und benutzen dazu eine Folge von logisch zusammengehörigen Operationen (z.B. UPDATE, DELETE, INSERT). Das bedeutet: Nach Ausführung einer einzelnen Operation kann die Datenbank inkonsistent sein. Das DBMS muß die Rückführung auf den bisher bekannten konsistenten Zustand ermöglichen. Alle Änderungen sind bis zu dieser definierten Stelle rückgängig zu machen.
SQL erkennt den Beginn einer Transaktion durch eine Datenmanipulation mit UPDATE, DELETE, INSERT. Zur Beendigung einer Transaktion gibt es die Befehle COMMIT und ROLLBACK

COMMIT
Mit diesem Befehl gibt der Anwender dem Datenbanksystem bekannt: Alle Operationen der Transaktion wurden richtig ausgeführt, die damit verbundenen Änderungen können auf der Datenbank dauerhaft gespeichert werden. Im interaktiven Modus unter SQL*PLUS führt ORACLE nach UPDATE, INSERT oder DELETE automatisch COMMIT durch.

Ein außerplanmäßiges Ende einer Applikation (z.B. Division durch 0) führt zum Ende der Transaktion, das automatisch mit ROLLBACK nachvollzogen wird. Weitere Endekriterien für eine Transaktion sind:

- der Aufruf von DDL-Kommandos
- Deadlocks führen auf ein automatisches COMMIT

- Beim normalen Ende der Arbeit mit SQL*PLUS, SQL*Forms oder einem dar anderen Tools wird der Benuzer aufgefordert, sich für COMMIT oder ROLLBACK zu entscheiden
- COMMIT und ROLLBACK bezeichnen jeweils das Ende einer Transaktion und den Beginn der nächsten

Hinweise:
- Mit dem SQL*PLUS-Befehl SET AUTOCOMMIT ON kann man vereinbaren, daß jede Änderung automatisch eingetragen wird, d.h. Nicht mehr zurückgenommen werden kann. Diese Vereinbarung wird mit SET AUTOCOMMIT OFF wieder aufgehoben
- ORACLE verfügt über die Befehle AUDIT und NOAUDIT. Sie bestimmen, welche Tabellenzugriffe in den Systemtabellen registriert werden sollen. AUDIT bestimmt den Anfang und NOAUDIT das Ende der Protokollierung.

Sperren von Tabellen

Parallelarbeit (Concurrency) und Konsistenz (Consistency) stellen widersprüchliche Anforderungen an das Datenbankverwaltungssystem. Ein gemeinsamer Datenbastand soll für die Verarbeitung an verschiedenen Orten und durch verschiedene Personen zur Verfügung stehen. Durch die fast gleichzeitige Änderung von Daten können Konflikte entstehen, die zu einer inkonsistenten (d.h. unbrauchbaren) Datenbasis führen. Die allgemeine Verfügbarkeit über Daten muß zumindest zeitweise eingeschränkt werden. ORACLE unterstützt das Sperren mit dem Befehl

```
LOCK TABLE Tabellenname [, .....]
IN {SHARE | SHARE UPDATE | EXCLUSIVE } MODE
[NOWAIT]
```

Jede Form einer Sperre erlischt mit dem Beginn der nächsten Transaktion. Jede Art von Beendigung von Transaktionen hebt gesetzte Sperren auf.
`Tabellenname`: steht für den Namen der zu sperrenden Tabelle

SHARE MODE
Dieser Modus verbietet anderen Anwendern:

- Die Anforderung von LOCKs im SHARE UPDATE MODE und im EXCLUSIVE MODE
- Jegliche Veränderung mit UPDATE, INSERT, DELETE und ALTER

SHARE MODE erlaubt anderen Anwendern:

- die Anwedndung von SELECT-Befehlen auf gesperrte Tabellen
- LOCKs im SHARE MODE zu setzen.

Mit der Option NOWAIT werden SQL-Anweisungen, die Änderungen ausführen, mit Meldung zurückgewiesen. Im Standardfall werden verändernde DML-Befehle akzeptiert, die Ausführung wird aber bis zur Aufhebung der bestehenden Sperre unterbunden.

EXCLUSIVE MODE
Dieser Modus verbietet:

- Die Anforderung jeglicher Art von Sperren auf die gleiche Tabelle
- jegliche Veränderung mit UPDATE, INSERT, DELETE, DROP und ALTER

Dieser Modus erlaubt anderen Anwendern die Anwendung von SELECT-Befehlen auf gesperrte Tabellen
SQL sperrt automatisch bei jedem der Befehle INSERT, UPDATE, DELETE die betroffenen Tabellen im EXCLUSIVE MODE.

SHARE UPDATE MODE
Dieser Modus untersagt anderen Anwendern

- die Anforderung von Sperren im SHARE oder EXCLUSIVE MODE
- die Veränderung derselben Zeile der Tabelle

Er erlaubt anderen Anwendern

- die gleichzeitige Anforderung von SHARE UPDATE Sperren
- die Veränderung anderer Zeilen derselben Tabelle

Hier wird nicht eine Tabelle gesperrt, sondern lediglich einige Zeilen.
Ein gesetzter LOCK bleibt bestehen, bis ein COMMIT oder ein ROLLBACK ausgeführt wird. Danach werden alle bestehenden LOCKs gelöscht. Oracle kontrolliert das Entstehen von Deadlocks automatisch und löst diese auf, indem einer der beteiligeten Befehle abgebrochen wird.

2.3.3 Rekursive und iterative Abfragen mit SQL

Berechnung der „transitiven Hülle“

SQL ist ausgerichtet an Relationenkalkül und Relationenalgebra. Bereits Anfang der 70er Jahre wurde erkannt, daß eine wichtige Klasse von Aufrufen, nämlich die Berechnung der sog. „transitiven Hülle“ nicht ausdrückbar ist, da Iterations- und Rekursionsmechanismen in SQL fehlen.

2.3.4 Einbindung von SQL in prozedurale Sprachen

SQL ist nicht nur eine interaktive Abfragesprache sondern auch eine Programmier-sprache für Datenbanken. Für den Einsatz als Datenbankprogrammiersprache fehlen aber Variablenkonzepte und die elementaren Konstrukte prozeduraler Programmiersprachen (z.B. REPEAT ... UNTIL, IF .. THEN ... ELSE, CASE ...). SQL muß daher in Programmiersprachen eingebunden werden.

2.3.4.1 Embedded SQL

Einzelsatzverarbeitung und das Cursorkonzept in SQL/92

Einzelsatzverarbeitung
Der „SINGLE-ROW-SELECT“ ermöglicht es, die Spalten eines einzelnen Datensatzes in Variablen einzulesen:

```
SELECT [ALL | DISTINCT] select-item-commalist
INTO targetvariable-commalist
From table-reference-commalist
[WHERE conditional-expression]
[GROUP BY column-ref-commalist]
[HAVING conditional-expression]
```

Mit UPDATE, INSERT und DELETE besteht die Möglichkeit Datensätze zu ändern. Ein Problem bei der Verwendung von SQL in Programmiersprachen der 3. Generation sind aber die Ergebnistabellen der SQL-Mengenoperationen (SELECT, JOIN, UNION, etc.). Auf Ergebnistabellen von SQL-Mengenoperationen kann mit Befehlen prozeduraler Sprachen nicht zugreifen. Abhilfe schafft das Cursorkonzept von SQL.

Das Cursorkonzept
Ein mit dem DECLARE-Befehl angelegter Cursor deklariert eine SQL-Mengenoperation.

```
DECLARE cursor [INSENSITIVE][SCROLL]
CURSOR FOR table-expression
[ORDER BY column [ASC | DESC] [, column [ASC | DESC] ] +]
[FOR {READ ONLY | UPDATE [OF column-commalist]}]
```

Nach dem Öffnen des Cursors (OPEN Cursor) kann man satzorientiert auf die Ergebnismenge des Cursors zugreifen. Der satzorientierte Zugriff erfolgt über FETCH-Befehle. Vom Ausgangspunkt des aktuellen Datensatzes kann der Programmierer über Navigationsfunktionen den Datensatzzeiger absolut oder relativ positionieren und mit der Deklaration des Cursor Zugriffsrechte (nur Lesen, Ändern, etc.) definieren. Das Positionieren des Cursor und das Übertragen der Variablen erfolgt über:

```
FETCH [[NEXT | PRIOR | FIRST | LAST | ABSOLUTE number | RELATIVE number]
FROM ] cursor
INTO parameter < variable-commalist
```

Zum Ändern / Löschen werden benutzt:

```
UPDATE table
 SET column = {value | DEFAULT | NULL} [, column = +]
  WHERE CURRENT OF cursor
DELETE Table
  WHERE CURRENT OF Cursor
```

Am Ende muß der Cursor geschlossen werden:

```
CLOSE cursor
```

Mit dem Cursorkonzept ist die Möglichkeit einer satzweisen Verarbeitung von Tabellen in prozeduralen Sprachen geschaffen. Notwendig sind jetzt noch Regeln nach denen SQL in prozeduralen Sprachen verwendet wird. Der SQL/92-Standard und Oracle/SQL bieten hier verschiedene Konzepte an.

Embedded SQL in Oracle

Jeder in einem Programm eingebettete SQL-Anweisung ist der Vorspann (Prefix) EXEC SQL zur Unterscheidung von Anweisungen der gastgebenden Sprache vorangestellt. Ein Semikolon bildet den Abschluß. Die Übersetzung der SQL-Anweisungen übernimmt ein Precompiler, die danach mit dem restlichen Quellcode übersetzt, gebunden und zum Ablauf gebracht werden kann.
Eingebettete SQL-Anweisungen können in zwei Gruppen aufgeteilt werden: ausführbar und deklarativ.

Ausführbare SQL-Anweisungen

Sie generieren aktuelle Aufrufe an die Datenbank. Zugelassen sind

- zur Datendefinition: ALTER, CREATE, DROP, RENAME
- zur Kontrolle des Datenzugriffs: CONNECT, GRANT, LOCK TABLE, REVOKE
- zur Datenmanipulation: DELETE, INSERT, UPDATE
- zur Datenwiedergewinnung: CLOSE, FETCH, OPEN, SELECT
- zur Verarbeitung von Transaktionen: COMMIT, ROLLBACK
- zur Anwendung von „dynamic SQL“: DESCRIBE, EXECUTE, PREPARE

Ausführbare Anweisungen können an allen Stellen im Programm angegeben werden, die auch für die Anweisungen der gastgebenden Sprache vorgesehen sind. Beziehen sich SQL-Anweisungen auf Variable der gastgebenden Sprache („host variables“), dann ist derartigen Referenzen zur Unterscheidung von SQL-Feldnamen ein „Doppelpunkt“ vorangestellt. „host variables“ werden in der DECLARE SECTION vereinbart.

Deklarative SQL-Anweisungen

Sie generieren keinen ausführbaren Code.

DECLARE SECTION

Die Deklaration von "host variables" für den Compiler der gastgebenden Sprache und für den SQL-Precompiler erfolgt in der DECLARE SECTION.

```
EXEC SQL BEGIN DECLARE SECTION
/*
 "host variables "
*/
EXEC SQL END DECLARE SECTION;
```

„host variables“ müssen einen Datentyp besitzen, der zu SQL-Datentypen kompatibel ist. In SQL-Anweisungen können nur deklarative Anweisungen verwendet werden. Es steht nur eine beschränkte Anzahl von Variablentypen zur Verfügung. Programmvariablen können den gleichen Namen wie Felder in Tabellen aufweisen. In SQL-Anweisungen steht vor Programmvariablen immer ein Doppelpunkt (`:variable`). Außerhalb von SQL-Anweisungen werden die Programmvariablen ohne Doppelpunkt angegeben.

SQLCA
ist eine Kontrollstruktur zur Steuerung und Überwachung der Kommunikation mit der Datenbank. In dieser Struktur werden Fehlermeldungen, Warnungen, Meldungstexte und diverse andere Informationen durch die Datenbank abgelegt.

```
struct sqlca {
             char sqlcaid[8];   // Enthaelt den String „sqlca“
             long sqlabc;       // Die Laenge der Struktur in Bytes
             long sqlcode;      // Variable für Fehlernummern zum
                                // zuletzt ausgefuehrten SQL-Statement
                                // 0 : Kein Fehler
                                // >0 : Ausnahme entdeckt, z.B. fetch
                                //   bzw. select geben „no rows“ zurück
                                // <0 : Die Anweisung wurde nicht ausge-
                                //      führt, ROLLBACK sollte folgen
             struct {
                     unsigned short sqlerrm1; // Laenge Meldungstext
                     char sqlerrmc[70];       // Meldungstext,
                                              //entspricht dem sqlcode
             } sqlerrm;
             char sqlerrp[8];        // wird nicht benutzt
             char sqlerrd[6];        // 6 Statuscodes
             char sqlwarn[8];        // 8 Warnungsflags
             char sqlext[8];         // nicht benutzt
};
```

Wichtige Informationen über den Programmablauf enthält „`sqlcode`“. Wird nach der Verarbeitung hier eine 0 zurückgegeben, wurde die Verarbeitung ordnungsgemäß ausgeführt: Positive Werte beziehen sich auf Warnungen (z.B. ORA-1403: no data found). Negative Werte weisen auf Fehler, der zugehörige Befehl wurde nicht ausgeführt.
Die SQLCA muß u.U. speziell über `EXEC SQL INCLUDE SQLCA` in das Programm einbezogen werden.

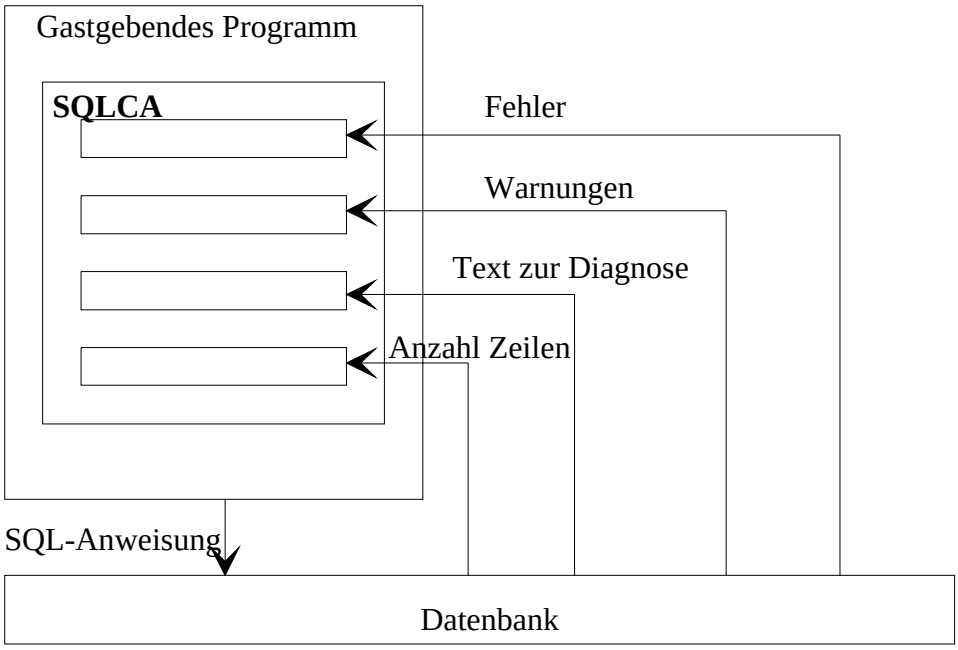


Abb. 2.3-5 : SQLCA

Exception Handling

Der Status nach ausgeführten SQL-Anweisungen kann auf zwei Wegen überprüft werden:

(1) Überprüfen der Komponentenwerte der SQLCA
(2) Automatische Fehlerüberwachung über die WHENEVER-Anweisung

Die vollständige WHENEVER-Anweisung ist festgelegt durch

```
EXEC SQL WHENEVER bedingung aktion
```

Falls dieses Kommando benutzt wird, dann wird die `SQLCA` automatisch auf die `bedingung` überprüft und gegebenenfalls die `aktion` ausgeführt. Die Bedingung kann sein:

`SQLERROR`: Ein Fehler ist aufgetreten, sqlcode hat einen negativen Wert.
`SQLWARNING`: Hier ist `sqlwarn[0]` gesetzt
`NOT FOUND`: Der `sqlcode` ist positiv, d.h.: Keine Zeile wurde gefunden, die die „where“-Bedingung erfüllt, bzw. „`select into`“ oder „`fetch`“ führen auf „`no rows`“.

„aktion“ kann sein

STOP : Das Programm endet mit einem `exit()`-Aufruf, alle SQL-Anweisungen, die noch nict abgeschlossen wurden, werden zurückgesetzt.
CONTINUE : Falls möglich, wird das Programm mit der folgenden Anweisung, die der fehlerhaften Anweisung folgt, fortgesetzt.
DO *funktion* : Das Programm verzweigt zu einer Fehlerbehandlungsfunktion *funktion*
GOTO *label* : Die Programmausführung verzweigt in das mit einem *label* markierte Statement.

Die Anweisungen können an beliebigen Programmstellen eingefügt werden.
Eine `WHENEVER`-Bedingung wird durch den Ausdruck `CONTINUE` annuliert:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

DECLARE-**Anweisung**
Vor der Ausführung einer in dem Programm eingebundenen Abfrage muß die in der Abfrage verwendete SQL-Anweisung deklariert werden

```
EXEC SQL DECLARE ang_cursor CURSOR FOR SELECT ...
```

Dadurch wird innerhalb des Programms ein Speicherbereich (Context Area) belegt, in dem das Ergebnis einer Abfrage gespeichert ist. „`ang_cursor`“ ist ein Bezeichner, der vom Precompiler benutzt wird und nicht in der DECARE SECTION definiert werden sollte. Ein **Cursor** verwaltet die SQL-Anweisung, hat Zugriff auf die Daten in der zugehörigen Context-Area und steuert die Verarbeitung. Es gibt 2 Cursor-Typen: **implizit** und **explizit**.
Implizit deklariert spricht der Cursor alle Datendefinitionen und Datenmanipulationen von SELECT-Anweisungen an, die nicht mehr als eine Zeile als Ergebnis zurückgeben. Die SQL-Anweisung kann ohne Cursor angegeben werden, falls bekannt ist: Die Abfrage liefert nur eine Zeile.
Explizit deklariert spricht ein Cursor Ergebnisse von SELECT-Anweisungen an, die mehr als eine Zeile umfassen. Diese Zeilen sind unter dem Namen „active set“ bekannt. Der explizit deklarierte Cursor verweist dann auf die aktuell verarbeiteten Zeilen (current row).
Erst nach der Bestimmung des Cursors kann die Abfrage ausgeführt werden. Zuerst wird der Cursor geöffnet, dann werden die Daten abgerufen:

```
EXEC SQL OPEN ang_cursor;
EXEC SQL FETCH ang_cursor INTO :variable1, :variable2;
```

Werden keine weiteren Daten benötigt oder wurde die letzte Programmzeile ermittelt, dann muß der Cursor wieder geschlossen werden:

```
EXEC SQL CLOSE ang_cursor;
```

Rückgabe einer Zeile
Bei eingebettetem SQL liefert ein `FETCH`-Befehl jeweils nur eine einzige Zeile. Nach Ausführung des OPEN-Befehls wird die erste Zeile zurückgegeben, die die in der Abfrage angegebene Bedingung erfüllt. Die nachfolgenden `FETCH`-Befehle ermitteln nacheinander, jeweils die nächste Zeile, die den Bedingungen entspricht. Werden weitere `FETCH`-Befehle nach Erhalt einer Fehlermeldung ausgeführt (z.B. "not found"), so liefert diese jedesmal diesselbe Fehlermeldung. Eine Abfrage kann wiederholt werden, indem der Cursor geschlossen, erneut geöffnet und dann der `FETCH`-Befehl nochmals aktiviert wird.
`WHERE`-Bedingungen können sich auf Programmvariable beziehen. Das bedeutet: Eine Abfrage wird beim Programmstart geöffnet und bleibt während des gesamten Programmablaufs geöffnet. Das Programm kann den Datensatz durch Veränderung der in den `WHERE`-Bedingungen verwendeten Variablenwerte gezielt auswählen. Ein Cursor kann ebenfals für die gesamte Programmdauer geöffnet werden, das Schließen muß erst bei Programmende erfolgen.
Die mögliche Anzahl gleichzeitig geöffneter Cursor richtet sich nach dem verwendeten Datenbankprodukt. Über mehrere Cursor können die von der Abfrage zurückgegebenen Werte unmittelbar zur Aktivierung anderer Abfragen verwendet werden.
Werden keine Zeilen ermittelt, die die gegebenen Bedingungen erfüllen, dann wird `sqlcode` gesetzt. Fehler werden ebenfalls über den `sqlcode` angezeigt. In

einem Programm wird über die Anweisung `WHENEVER NOT FOUND` der Fehler erkannt. Falls die Ergebnismenge leer ist, gibt `FETCH` „`no data found`“ zurück und markiert dies über den `sqlcode`.

2.3.4.1.1 Embedded SQL in Oracle mit dem Precompiler Pro*C

`Pro*C` ist ein Werkzeug, daß SQL-Anweisungen in einem C-Quellcode-Programm in Oracle-Datenbankanweisungen umsetzen kann.

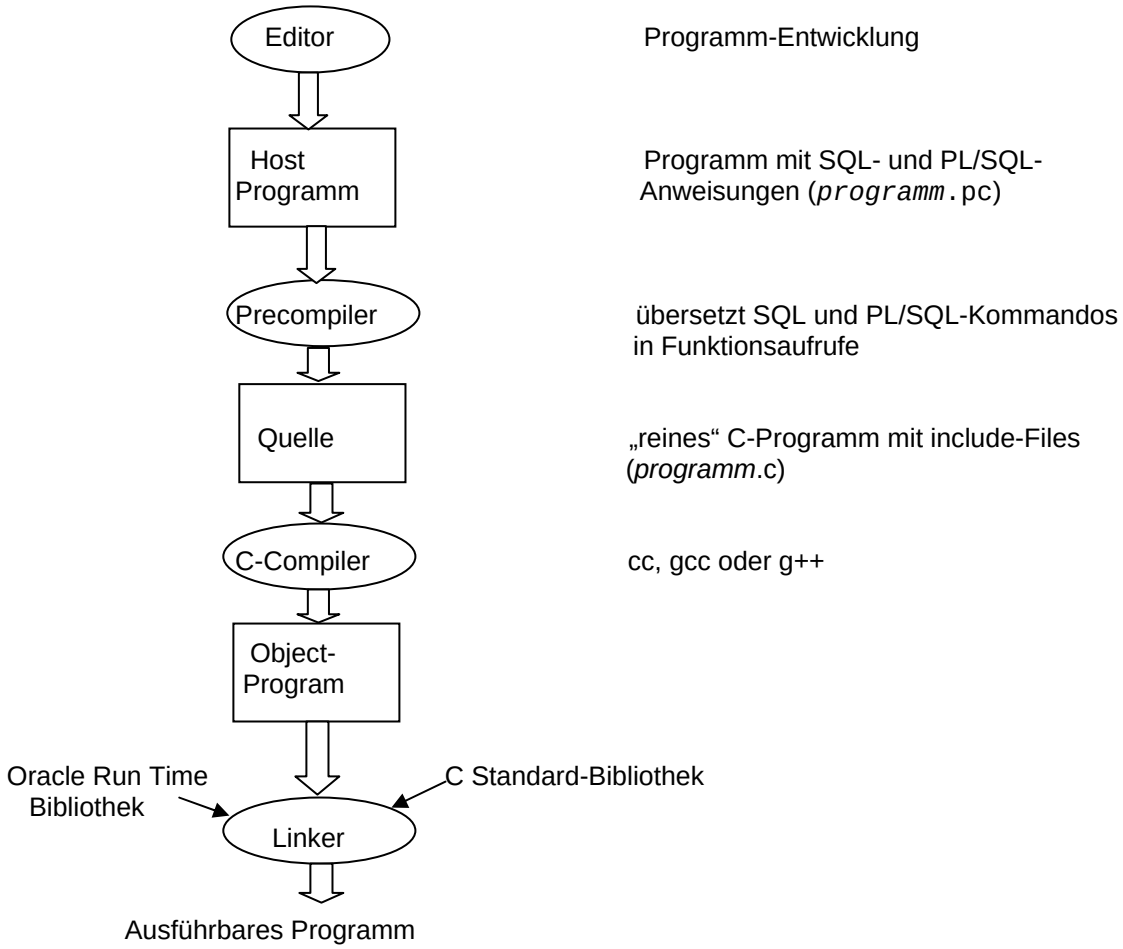


Abb.: Übersetzung eines Pro*C-Programms

Ein `Pro*C` - Programm besteht aus 2 Teilen:

- dem Prolog zur Anwendung (Application Prologue)
- dem eigentlichen Anwendungprogramm (Application Body)

a) Application Prologue

Dieser Vorspann besteht aus 3 Teilen:

1. Der „`DECLARE` Section“ zur Vereinbarung von Variablen

- „Host Variables“ können folgende Datentypen annehmen:
  `char` *`name`* /* einzelnes Zeichen */
  `char` *`name`*`[n]` /* Array mit n Zeichen */

`float` /* floating point */
VARCHAR *name*`[n]` /* Zeichenkette in variabler Länge */
VARCHAR wird vom `Pro*C` Precompiler in eine „`structure`“ mit einem n Bytes großen Zeichenfeld und einem zwei Bytes großen Längenfeld konvertiert.

- „Host Variables“ werden in SQL und PL/SQL über ein vorangestelltes Zeichen „`:`“ referenziert.
- Nur eindimensionale Felder mit einfachen C-Typen werden unterstützt
- Zweidimensionale Felder sind nur erlaubt für `char[m][n]` bzw. `VARCHAR[m][n]`. „m“ bestimmt die Anzahl der Zeichenketten und „n“ die maximale Länge der Zeichenketten
- Zeiger auf einfache C-Typen werden unterstützt. Zeiger auf `char[n]` bzw. `VARCHAR[n]` sollten als Zeiger auf `char` bzw. VARCHAR vereinbart sein
- Zeiger auf „Felder“ (array of pointers) sind nicht zugelassen.
- Aus Kompatibilitätsgründen mit dem ANSI-Standard ist die Deklaration „`extern char[n]`“ ohne Längenangabe zugelassen, z.B.:

```
EXEC SQL BEGIN DECLARE SECTION;
...
extern char nachricht[];
...
EXEC SQL END DECLARE SECTION;
```

2. Der Anweisung „`INCLUDE SQLCA`“ zur Referenz auf den SQL-Kommunikationsbereich.

3. der Anweisung `CONNECT` zur Verbindung mit der Oracle-Datenbank.

`EXEC SQL CONNECT :`*`userid`* `IDENTIFIED BY :`*`passwort`*

`userid` und *`passwort`* sind „`host variables`“ vom Typ VARCHAR.

Oracle kennt folgende interne (kompatible) Datentypen

Interner Typ	C-Type	Beschreibung
CHAR(X)[166]	char	einzelnes Zeichen
VARCHAR(X)	char[n] VARCHAR[n] int short long float double	n Bytes umfassendes Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge integer small integer large integer floating point number double-precission floating point number
NUMBER	int	integer
NUMBER(P,S)[167]	short long float double char char[n]	small integer large integer floeting-point number double-precision floating-point number einzelnen Zeichen n Bytes umfassendes Zeichenfeld[168]
DATE[169]	char[n] VARCHAR[n]	n Bytes umfassendes Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge
LONG	char[n] VARCHAR[n]	n Bytes umfassendes Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge

[166] X umfaßt einen Bereich von 1 bis 255, 1 ist Default-Wert

[167] P umfaßt einen bereich von 2 bis 38, S von -84 bis 127

[168] Zeichenketten können nach NUMBER konvertiert werden, falls sie konvertierbare Zahlen enthalten ('0' bis '9','.',',','+','-','E','e')

[169] Im Default-Format (DD-MON-YY) erfordert ein Zeichenkettentyp 9 Zeichen. Wird er als binärere Typ konvertiert, dann erfordert er 7 Zeichen

RAW(X)	unsigned char[n] VARCHAR[n]	n Bytes umfassendes vorzeichenloses Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge
LONG RAW	unsigned char[n] VARCHAR[n]	n Bytes umfassendes vorzeichenloses Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge
ROWID[170]	char[n] VARCHAR[n]	n Bytes umfassendes Zeichenfeld n Bytes umfassendes Zeichenfeld variabler Länge

b) Application Body

Hier befinden sich die SQL-Anweisungen. Häufig brauchen SQL-Anweisungen zusätzliche Befehlsfolgen.

Deklaration und Bearbeitung eines SQL-Cursor

Nicht jede SELECT-Anweisung liefert automatisch nur eine einzige Zeile als Ergebnis zurück. Ein Cursor stellt eine ganze Ergenismenge zur Bearbeitung bereit.
Folgende Befehle ermöglichen die Bearbeitung eines Cursor:

SQL-Befehl	Beschreibung
DECARE CURSOR	Deklaration eines Cursor
OPEN	Öffnet einen SQL-Cursor und setzt diesen vor die erste Zeile der Resultat-Tabelle
FETCH	Setzt den Cursor auf die nächste Zeile und gibt diesen Zeileninhalt aus
CLOSE	Schließt den geöffnetet SQL-Cursor

Abb.: SQL-Cursor-Befehle

Ein Cursor wird über DECLARE deklariert:

```
DECARE Cursorname CURSOR FOR
Select-Anweisung
[FOR {READ ONLY | UPDATE [OF Spalte [,...] ] } ]
```

Bsp.:

```
EXEC SQL DECLARE C1 CURSOR FOR
      SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME,
        TO_CHAR(ANGESTELLTE.GEBJAHR,'dd.mm.yy')
      FROM ANGESTELLTE, ABTEILUNG
      WHERE ANGESTELLTE.ABT_ID    = ABTEILUNG.ABT_ID  AND
         ABTEILUNG.BEZEICHNUNG = 'KONSTRUKTION';
```

Standarmäßig gilt für jeden Cursor die Einstellung FOR UPDATE.
Ein Cursor muß vor der Bearbeitung geöffnet werden:

```
EXEC SQL OPEN Cursorname
```

Das Schließen funktioniert analog:

EXEC SQL CLOSE *Cursorname*

[170] Falls die Konvertierung in eine zeichenkette erfolgt, erfordert ROWID 18 Bytes. Bei einer Konvertierung als Binärwert ist die Länge systemabhängig

Ein nicht explizit geschlossener Cursor bleibt geöffnet, bis die Anwendung beendet wird. Die Datenbank kann auf eine maximale Anzahl geöffneter Cursor beschränkt sein.
Ein einzelner Datensatz aus der Ergebnismenge wird mit FETCH geholt:

```
FETCH [FROM] Cursorname INTO Variablenliste;
```

Bei der Variablenliste handelt es sich um durch Kommas getrennte Hostvariablen (mit dem typischen Doppelpunkt davor). Die Reihenfolge der Variablen muß der Reihenfolge der im Cursor befindlichen Attribute entsprechen.
Eine Änderung am Datenbestand in Abhängigkeit vom aktuell gültigen Tupel im Cursor kann durch zwei Anweisungen erreicht werden:

DELETE FROM { Tabellenname | Viewname }
WHERE CURRENT OF Cursorname

Diese Anweisung löscht aus der angebenen tabelle oder Sicht das Tupel, das im Cursor gerade aktuell angezeigt wird.

UPDATE { Tabellenname | Viewname }
SET { Spalte = NeuerWert } [, ...]
WHERE CURRENT OF Cursorname;

Diese Anweisung aktualisiert in der Tabelle oder View den Spaltenwert desjenigen Tupels, das im Cursor gerade aktiv ist.

Abfrage von SQL-Fehlern

Die Beispielanwendung verwendet anstelle von SQLSTATE den oraclespezifischen SQLCODE, der aus der `sqlca` ausgelesen wird.

Code	Ursache
0	Erfolgreiche Beendigung
100	Daten nicht gefunden
<0	Fehler

Abb.: SQLCODE-Fehlercode

Die Bearbeitung dieser numersichen werte ist wesentlich einfacher als die Verarbeitung von SQLSTATE, der als Zeichenkette zurückgeliefert wird:

Code	Ursache
‘00‘	Erfogreiche Beendigung
‘01‘	Warnung
‘02‘	Daten nicht gefunden
‘08‘	Verbindungsaufbau-Fehler
‘0A‘	Merkmal wird nicht unterstützt
‘22‘	Datenfehler (z.B. Division durch 0)
‘23‘	(Tabellen/Spaten-)Bedingung ist verletzt
‘2A‘	SQL-Syntax- oder Zugriffsfehler
‘2D‘	Nichterlaubte Transaktionsbeendigung
‘34‘	Ungültiger Cursorname
‘3D‘	Ungültiger Katalogname
‘3F‘	Ungültiger Schemaname
‘40‘	Rollback
‘42‘	Syntax- oder Zugriffsfehler
‘44‘	Check-Bedingung ist verletzt

Abb.: Fehlerwerte von SQLSTATE

In Abhängigkeit von der Anwendung kann auf einen Fehler mit

EXEC SQL COMMIT [WORK]
oder
EXEC SQL ROLLBACK [WORK]

reagiert werden.
Die Klausel

```
WHENEVER { SQLERROR | NOT FOUND } { CONTINUE | GOTO Label };
```

wird immer dann gültig, wenn ein SQL-Fehler jeglicher Art registriert wird.

Ein C-Programm mit „Embedded-SQL“

```
/*=======================================================*/
/*                                                       */
/* Das Programm ermittelt Loesungen fuer die SQL-        */
/* Abfrage:                                              */
/* Ermittle alle Angestellten, die in der Abteilung      */
/* 'Konstruktion' beschaeftigt sind.                     */
/*                                                       */
/* SELECT ANGESTELLTE.ID, ANGESTELLTE.NAME,              */
/*        TO_CHAR(ANGESTELLTE.GEBJAHR,'dd.mm.yy')        */
/*    FROM ANGESTELLTE, ABTEILUNG                        */
/*    WHERE ANGESTELLTE.ABT_ID=ABTEILUNG.ID AND          */
/*          ABTEILUNG.BEZ='KONSTRUKTION';                */
/*                                                       */
/*=======================================================*/

#include <stdio.h>

/*=======================================================*/
/* In der DECLARE SECTION muessen alle Variablen         */
/* deklariert werden, die von einem der SQL-Makros       */
/* benutzt werden sollen                                 */
/*                                                        */
/* Mode ANSI : C-Character-Strings mit schliessendem     */
/*             0-Byte koennen ohne Aenderung benutzt     */
/*             werden fuer SQL-Aufrufe                   */
/*                                                       */
/*=======================================================*/

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR  uid     [20];
   VARCHAR  pwd     [20];
   VARCHAR anggeb   [ 8];
   VARCHAR angid    [ 4];
   VARCHAR angname  [11];
   VARCHAR angabt   [ 4];
   VARCHAR abtid    [ 3];
   VARCHAR abtbez   [41];
   VARCHAR db_string[20];

EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca;
void fehler();
```

```
/* Beginn des Hauptprogrammes */

main (int argi, char * argv [])
{
   strcpy ( uid.arr, "xyz12345" );  /* ORACLE-USERID   */
   uid.len = strlen ( uid.arr );
   strcpy ( pwd.arr, "xyz12345" );  /* ORACLE-Passwort */
   pwd.len = strlen ( pwd.arr );
   strcpy(db_string.arr,"rfhs1012_ora2");
   db_string.len = strlen(db_string.arr);

   /*=================================================*/
   /* Definition eines Fehlerausgangs fuer den Fall,    */
   /* dass ein Fehler bei der Ausfuehrung eines SQL-    */
   /* Kommandos passiert                              */
   /*=================================================*/

   EXEC SQL WHENEVER SQLERROR DO fehler();

   /*=================================================*/
   /* Einloggen in das ORACLE-System unter Benutzung    */
   /* von USERID und Passwort                         */
   /*=================================================*/

EXEC SQL CONNECT :uid IDENTIFIED BY :pwd
         USING :db_string;

   printf ( "Connected to ORACLE as user '%s' \n", uid.arr );

   /*=================================================*/
   /* Deklaration eines CURSORS fuer das gewuenschte    */
   /* SQL-Kommando                                    */
   /*                                                 */
   /* Ein CURSOR ist ein Arbeitsbereich, der von ORACLE */
   /* Pro*C benutzt wird, und der die Ergebnisse einer  */
   /* SQL-Abfrage enthaelt.                           */
   /*                                                 */
   /* Der CURSOR wird mit dem folgenden Kommando mit    */
   /* einer SQL-Abfrage in Beziehung gebracht.        */
   /*                                                 */
   /*=================================================*/

   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME,
        TO_CHAR(ANGESTELLTE.GEBJAHR,'dd.mm.yy')
      FROM ANGESTELLTE, ABTEILUNG
      WHERE ANGESTELLTE.ABT_ID    = ABTEILUNG.ABT_ID  AND
         ABTEILUNG.BEZEICHNUNG = 'KONSTRUKTION';


   /*=================================================*/
   /* Der zuvor definierte CURSOR wird eroeffnet.       */
   /* Die mit ihm assoziierte Abfrage wird ausgefuehrt  */
   /* und alle gefundenen Loesungen in der einer        */
   /* Tabelle im Arbeitsbereich abgespeichert.        */
   /* Der CURSOR wird auf die erste Loesung positioniert*/
   /*=================================================*/

   EXEC SQL OPEN C1;

   /*=====================================================*/
   /*    Definition der Aktion, die durchgefuehrt werden   */
   /*    soll, falls alle Loesungen der Abfrage abge-      */
   /*    arbeitet worden sind.                           */
   /*=====================================================*/
```

```
   EXEC SQL WHENEVER NOT FOUND STOP;

   /* Schleife ueber alle gefundenen Loesungen */

   for ( ; ; )
   {

      /*=============================================*/
      /* Die naechste (erste) Loesung wird ermittelt    */
      /* und die gewuenschten Felder in die angebotenen */
      /* Variablen kopiert                           */
      /*=============================================*/

      EXEC SQL FETCH C1
          INTO :angid, :angname, :anggeb;

      /* Ausgabe der aktuellen Loesung */

      printf ("%s %s %s\n", angid.arr, angname.arr, anggeb.arr);

   }

}
   /* Fehlerausgang */

   void fehler()
   {

   /* Ausgabe der ORACLE-Fehlermeldung */

   printf ("Fehler aufgetreten\n");
   printf ("%d\n", sqlca.sqlcode);
   exit (-1);
   }
```

Ein C++-Programm mit Embedded SQL

2.3.4.1.2 Embedded SQL mit Java (SQLJ in Oracle)

Mit dem gemeinsamen Standard **SQLJ** wollen die RDBMS-Hersteller IBM, Informix, Oracle und Sybase Schwierigkeiten mit der JDBC-Schnittstelle beheben. Der SQLJ-Standard ist in drei Teile gegliedert und berücksichtigt folgende Aspekte des Datenbankzugriffs aus Java-Anwendungen:

- Embedded SQL mit Java
- Stored Procedures und benutzerdefinierte Funktionen mit Java
- Java-Klassen als benutzerdefinierte SQL-Datentypen

SQLJ[171] Einführung: Einbettungsprinzip

- Einbettung von statischen SQL-Anweisungen direkt in den Java-Quelltext
- Statisch bedeutet: Die Anweisungen sind zur Übersetzungszeit definiert und können während der Abarbeitung des Programms nicht geändert werden.
- Vorübersetzung des erweiterten Quelltextes in "echten" Java Code durch den Präcompiler (Translator)

Übersetzen: Quelldateien mit der Extension `.sqlj` werden dem SQLJ-Übersetzer übergeben. Dieser sucht nach den Zeilen mit `#sql` ab und ersetzt sie durch gültigen Java-Code, der mit Hilfe von JDBC die notwendigen Operationen durchführt. (SQLJ definiert keine eigene Datenbankschnittstelle, sondern ist nur als Aufsatz auf die bereits bestehende JDBC-API zu sehen, die Verbindung zur Datenbank erfolgt mit dem verfügbaren JDBC-Treiber.) Darüber hinaus kann der SQLJ-Übersetzer die Syntax der eingebetteten SQL-Anweisungen überprüfen und bereits beim Kompilieren Aussagen über fehlerhafte SQL-Anweisungen machen. Prinzipiell besteht auch die Möglichkeit eine Verbindung zur Darstellung während der Übersetzung herzustellen, damit Namen der Tabellen und Spalten in SQL-Anweisungen überprüft werden können.
Der SQLJ-Translator

- übersetzt ein gegebenes SQLJ-Programm in ein Java Programm ohne SQL-Klauseln. Die SQLJ-Quelltexte werden mit dem Kommandozeilenprogramm `sqlj` kompiliert.
- es werden SQLJ-Profile angelegt, die Informationen über die eingebetteten SQL-Operationen beinhalten.

```
sqlj optionen dateien
```

optionen: Eine durch Leerzeilen getrennte Liste von Parametern
dateien: Eine oder mehrere Dateien, die kompiliert werden sollen. Der Platzhalter * kann bei den Dateien verwendet werden.
Der SQLJ-Translator kann die Syntax der SQL-Anweisungen direkt beim Kompilieren überprüfen. Dazu setzt man bspw. den folgenden Kommandozeilenbefehl ab:

```
sqlj –user=xyz12345/xyz12345 –url= jdbc:oracle:thin:@rfhs8012:1523:ora10g
```

Nach dem Kompilieren übersetzt ein Java-Compiler die generierten Java-Dateien in Java-Byte-Code.

[171] Der SQLJ-Standard wurde von Compaq, IBM, Informix, Micro Focus, Micosoft, Oracle, Sun und Sybase entwickelt

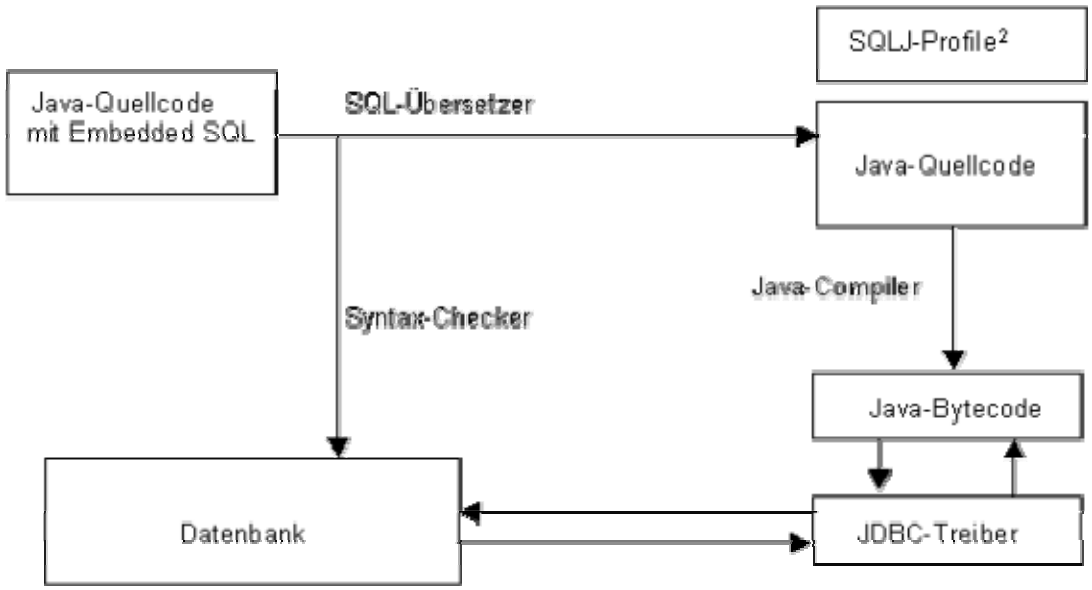


Abb.1 : Schematischer Ablauf beim Einbinden von Embedded SQL in Java-Anwendungen

Festlegen der Verbindungsdaten mit der Klasse Oracle. Über die Klasse `oracle.sqlj.runtime.Oracle` kann ein Verbindungskontext initialisiert werden. Dazu werden die Verbindungsdaten in einer externen Datei ("connect.properties") im Property-Format von Java festgelegt.

```
sqlj.url=jdbc:oracle:thin:@rfhs8012:1523:ora10g
sqlj.user=xyz12345
sqlj.password=xyz12345
```

Damit die SQLJ-Umgebung diese Verbindungsdaten nutzt, ist im Java-Quelltext die Methode `connect()` der Klasse `oracle.sqlj.runtime.Oracle` aufzurufen: `void connect(Class class, String propertyFile)`.
In dem folgenden Quelltext wird über die Klasse Oracle eine einfache Verbindung zur Datenbank hergestellt

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;

public class HalloWelt
{
 public static void main(String args[]) throws Exception
 {
  java.sql.Date current_date;
  try
  {
   // Verbindung zur Datenbank
   Oracle.connect(HalloWelt.class,"connect.properties");
   // Bestimme das aktuelle Datum aus der Datenbank
   #sql { SELECT sysdate INTO :current_date FROM dual };

   // Ergebnis ausgeben
   System.out.println("Hallo Welt am " + current_date);
  }
  catch (SQLException e)
  { System.out.println("SQLException " + e); }
  finally {
           try { Oracle.close(); }
           catch (SQLException e)
           { System.err.println("SQLException " + e); }
```

```
        }
 }
}
```

Verbindungskontexte mit `DefaultContext`. Einen Verbindungskontext, der innerhalb eines SQLJ-Blocks genutzt werden kann, erstellt man über die Klasse `sqlj.runtime.refDefaultContext`. Einen DefaultContext legt man mit Hilfe einer gewöhnlichen JDBC-Verbindung, also mit einem Objekt vom Typ `java.sql.Connection` über folgenden Konstruktor an: `DefaultContext(java.sql.Connection` *`conn`*) (*`conn`*: Die Datenbankverbindung, die durch diesen Kontext verwendet werden soll). Alternativ können dem Konstruktor auch die Verebindungsdaten übergeben werden. Die Klasse Defaultkontext stellt dann intern eine Verbindung anhand der übergebenen Daten her:
`DefaultContext(java.lang.String` *`url`*`, java.lang.String` *`user`*`, java.lang.String` *`password`*`, boolean autoCommit`[172]`)`
Mit der statischen Methode `setDefaultContext()` wird der Defaultkontext bei der SQLJ-Implementierung als standardmäßig genutzte Verbindung registriert: `DefaultContext.setDefaultContext(DefaultContext` *`context`*`)`. Alle darauf folgenden SQLJ-Blöcke nutzen dann die im Parameter *`context`* gekapselte Verbindung.
Wurden mehrere Verbindungskontexte angelegt, dann kann bei einem SQLJ-Block festgelegt werden, welchen Kontext dieser verwenden soll: `#sql [`*`Kontextname`*`]` *`SQLJ-Anweisung`*

Ausführen und Auswerten über Anfragen. Ein SQLJ-Block wird grundsätzlich als Anweisung interpretiert, die über die zuvor definierte SQLJ-Anweisung ausgeführt wird.

`#sql { `*`SQL-Anweisung`*` };`
`SQL-Anweisung`: eine beliebige, gültige SQL-Anweisung

Eine Abfrage mit mehreren Datensätzen als Ergebnis erfordert in der Regel eine Schleife und spezielle Methoden der Auswertung der Ergebnismenge. In der JDBC kann man über ResultSet ein solches mehrere Zeilen umfassendes Ergebnis gut auswerten. SQLJ definiert dazu so genannte Iteratoren.
Abfrage der Anzahl der geänderten Datensätze: Über die Klasse ExecutionContext, die nach einer SQLJ-Anweisung aus der DML implizit intialisiert wird, kann festgestellt werden, wie viele Datensätze von einer Änderung betroffen sind. Eine Referenz auf den aktuellen ExecutionContext erhält man von der Klasse `DefaultContext`, und zwar über deren Methode `getExecutionContext`. Die Anzahl der geänderten Datensätze erhält man dort über die Methode `getUpdateCount`.

Datenaustausch über Host-Variablen: Der Datenaustausch zwischen SQL- und Java-Code erfolgt über Host-Variablen (und Iteratoren). Host-Variablen bieten Zugriff auf Ergebnisse von Select-Anweisungen, die nur eine Zeile als Ergebnis liefern, darüber hinaus können sie Parameter für beliebig andere SQL-Anweisungen setzen. Es handelt sich um Variablen der jeweiligen Host-Sprache, (d.h. Java), die in SQL-Anweisungen auftreten können. In SQL-Anweisungen

[172] Falls alle SQL-Anweisungen direkt als eine isolierte Transaktion ausgeführt werden sollen, dann ist `true` zu übergeben. Falls die Transaktionssteuerung übernommen werden soll, ist `false` zu übergeben.

werden Host-Variablen durch eine Doppelpunkt >>:<< gekennzeichnet. Host-Variablen können in beiden Richtungen des Datenaustauschs eingesetzt werden, d.h. zur Übergabe von Java-Werten an:

- die SQL-Anweisung: (IN), Standardform
- oder als Ergebnisse: OUT
- bzw. in beide Richtungen gleichzeitig: (INOUT)

```
#sql { INSERT INTO kunden
       VALUES (:IN (++kundenNR), :IN nachname, :IN sysdate) };
```

Eie SQL-Klausel kann auch Werte zurückliefern, wie z.B. beim Aufruf einer gespeicherten Funktion:
`#sql` *`Variable`* `= { VALUES(` *`Funktionsaufruf`* `};`
`Variable`: Eine Hostvariable passenden Typs, die den Rückgabewert aufnehmen soll.
`Funktionsaufruf`: Der Aufruf der gespeicherten Funktion mit Parametern, etc.
Bsp.: Abfrage des aktuellen Datums auf dem Oracle-Server über die Funktion SYSDATE.

```
java.sql.Date datum;
#sql datum = { VALUES(SYSDATE)) };
System.out.println(datum);
```

Gespeicherte Prozeduren werden in SQLJ durch das vorangestellte Schlüsselwort CALL aufgerufen werden: `#sql { CALL` *`proc`*`(`*`Parameterliste`*`) };`. Diesen Prozeduren können IN-, OUT-, INOUT-Parameter übergeben werden.

Datenaustausch über Iteratoren: beim Zugriff auf mehrzeilige Ergebnismengen kommen Iteratoren zum Einsatz[173]. SQLJ bietet zwei Arten von Iteratoren an: benannte Iteratoren (Named Iterator mit Zugriff auf das Ergebnis über Methoden, die die Namen der Tabellenspalten tragen) und Positions-Iteratoren. Ein Iterator definiert man allgemein über einen SQLJ-Block in folgendem Format:

`#sql [`*`modifikatoren`*`] iterator` *`iteratorname`* `[implements` *`interface`*`] (`*`parameterliste`*`)`

Alle von der SQLJ-Implementierung generierten Iteratoren erfüllen die Vorgaben der Schnittstelle `sqlj.runtime.ResultSetIterator`. Programmierer haben u.a. Zugriff auf folgenden Methoden:

`close()`
schließt einen Iterator bzw. die darunter liegende Ergebnismenge
`isClosed()`
Die Methode liefert true, falls die darunterliegende Ergenismenge geschlossen wurde.
`getResultSet()`
Über diese Methode kann man eine Referenz auf das dem Iterator zugrundeliegende Abfrageergebnis in Form eines java.sql.ResultSet erhalten.
`next()`
Der Cursor der Ergebnismenge wechselt zum nächsten Datensatz. Die Methode liefert `true`, wenn ein weiterer Datensatz in der Ergebnismenge vorhanden ist, bzw. `false`, wenn der Iterator bereits beim letzten Datensatz angelegt ist.
`endFetch()`
Diese Methode liefert true, nachdem die letzte Zeile erreicht wurde.

[173] Die Verarbeitung von Anfrageergebnissen mit Hilfe von Host-Ausdrücken ist nur auf wenige SQL-Konstrukte beschränkt, wie die SELECT ... INTO Klausel oder den Aufruf von gespeicherten Prozeduren.

Benannte. Iteratoren. Der Iterator wird in einem SQLJ-Block deklariert. Nach dem Namen des Iterators (z.B. `angestellte`) folgt eine Liste der über den Iterator abzufragenden Attribute, inklusive des bevorzugten Datentyps. Der SQLJ-Translator generiert beim Kompilieren eine Java-Klasse, die die Vorgaben der Deklaration und der im Paket sqlj.runtime definierten Interfaces erfüllt. Ein benannter Iterator definiert seine Parameter namentlich
Die Attributnamen des Iterators müssen identisch mit den Namen der Spalten des Abfrage-Ergebnisses sein. Die Reihenfolge der Spalten bei der Abfrage oder bei der Iterator-Deklaration ist egal, die Verknüpfung erfolgt nur über die Namen.
Bsp.: `AngSchemaDemo.sqlj`

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

#sql iterator Angestellte (String name, String abt_id);

class AngSchemaDemo
{
  public static void main(String[] args) throws SQLException
  {

    Oracle.connect(AngSchemaDemo.class, "connect.properties");

    Angestellte ang;

    #sql ang = { SELECT name, abt_id FROM angestellte };

    while (ang.next())
    {
     System.out.println("Angestellte: " + ang.name() + " (" +
                          ang.abt_id() + ")");
    }
  }
}
```

Positions-Iteratoren. Die Definition von Positions-Iteratoren legt nur die Datentypen fest, nicht jedoch die Namen. Die Reihenfolge bei der Definition des Iterators muß in diesem Fall mit der Spaltenreihenfolge in der SQL-Abfrage übereinstimmen. Unbenannte Iteratoren werden durch ihre Position bestimmt. Der Datenzugriff weicht auch vom benannten Iterator ab. Via Fetch-Anweisung lassen sich die aktuellen Werte des Iterators in Host-Variablen einlesen: `#SQL { FETCH :positer INTO : ..., : ... };`

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

// Deklaration des Iterators

#sql iterator AngIter (String, String, String);

class AngPosIterDemo
{
  public static void main(String[] args) throws SQLException
  {
    // Connect herstellen
    Oracle.connect(AngPosIterDemo.class, "connect.properties");
    // Host-Variable
    String angID       = "";;
    String angName     = "";;
    String angGebdatum = "";
    // Variable auf Iterator definieren
```

```
    AngIter angPosIter;
    // Abfrage durchfuehren
    #sql angPosIter = { SELECT ang_id, name, gebdatum FROM angestellte };
    // Ergebnis ausgeben
    while (true)
    {
     #sql { FETCH :angPosIter INTO :angID, :angName, :angGebdatum };
     if (angPosIter.endFetch()) break;
     System.out.println(angID + ", " + angName + ", " + angGebdatum);
    }
    angPosIter.close();
  }
}
```

Iteratoren über eigenes Interface verwenden. Die Syntax der Iterator-Deklaration ermöglicht es, über das Schlüsselwort interface ein eigenes Java-Interface für den Iterator zu verwenden.

Kontexte: Eine Verbindung wird in SQLJ grundsätzlich durch einen Verbindungskontext, eine Instanz der Klasse `sqlj.runtime.ConnectionContext` repräsentiert. Der Verbindungskontext spezifiziert die Datenbank mit den assoziierten Schemata und die Verbindungsinformationen. Die Verbindungsinformationen bestehen aus: Benutzername, Passwort, Auto-Commit-Modus: `#SQL [`*`Kontextname`*`] `*`SQLJ-Anweisung.`*
Kontextname: Der Name der Variablen des Verbindungskontextes im Java-Quelltext
SQLJ-Anweisung: Ein gültige SQLJ-Anweisung
Bsp.: Verwendung verschiedener Verbindungskontexte in einem SQLJ-Programm

Transaktionssteuerung mit SQLJ. Eine Transaktion ist eine Folge von SQL- und PL/SQL-Anweisungen, die nach dem Prinzip alles oder nichts ausgeführt werden, d.h. Falls eine der Anweisungen scheitert, die Transaktion also nicht ordentlich beendet wird, werden alle Anweisungen dieser Transaktion wieder rückgängig gemacht.
Eine Transaktion beginnt automatisch mit dem ersten SQLJ-Block, der eine SQL-Anweisung an die Datenbank sendet, und endet mit einem SQLJ-Block mit der Anweisung COMMIT (`#sql { COMMIT };`). Falls nach einem Abbruch einer Transaktion alle Änderungen, die bisher in der Transaktion vorgenommen wurden, widerrufen werden sollen, geschieht dies über ROLLBACK (`#sql { ROLLBACK };`)

Transaktionssicherheit festlegen. Die Einstellungen zur Transaktionssicherheit legen fest, wie parallel ablaufende Transaktionen voneinander "isoliert" werden: `#sql { SET TRANSACTION [` *`Zugriffsmodus`* `] [,] [ISOLATION LEVEL` *`Isolationsstufe`* `] };`
Zugriffsmodus: `read only` oder `read write`. Legt fest, welche Art von SQL-Anweisungen in einer Transaktion erlaubt sind. Standardmäßig ist auf `read write` eingestellt.
`Isolationsstufe`: `read committed` oder `serializable`. Oracle nutzt standardmäßig die Einstellung `read committed`. Während einer Transaktion können dann allerdings nicht wiederholbare Lesevorgänge auftreten, d.h.: Falls in einer längeren Transaktion zweimal die gleiche SELECT-Anweisung ausgeführt wurde, und die betroffenen Datensätze in der Zwischenzeit von einer anderen

Transaktion verändert wurden, dann erhält man zwei verschiedenen Ergebnisse. Das erste Ergebnis ist also nocht wiederholbar. Die Isolationsstufe `serializable` sorgt hingegen durch entsprechende Sperren und Zwischenspeicher dafür, dass man die Daten nur in dem Zustand erhält, der am Anfang der Transaktion galt.

Bsp.: Experimente mit Einstellungen zur Transaktionssicherheit

```
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;

public class ExpTransSicher
{
  public static void main(String args[])
  {
    try {
         DefaultContext trans1 = Oracle.connect(ExpTransSicher.class,
         "connect.properties");
         DefaultContext trans2 = new DefaultContext(
         "jdbc:oracle:thin:@rfhs8012:1523:ora10g",
         "saj39122","saj39122",
         false);
         DefaultContext.setDefaultContext(trans1);
         /* Der folgende Block ist nur dann zu aktivieren, falls der
            nicht wiederholbare Lesevorgang vermieden werden soll
         #sql [trans1] {
           SET TRANSACTION read write
           ISOLATION LEVEL serializable
                       };
         */
         String vorname = null, phantom = null;
         // Den Namen erstmals lesen
         #sql [trans1] {
                        SELECT name INTO :vorname
                        FROM angestellte
                        WHERE ang_id = 'A1'
                       };
         System.out.println("[trans1]: Name '" + vorname + "'");
         // Veraenderung vom Namen in einer 2. Verbindung
         #sql [trans2] {
                        UPDATE angestellte
                        SET name = 'Phantom' WHERE ang_id = 'A1'
                       };
         // Die Transaktion beenden
         #sql [trans2] { COMMIT };
         // Mit trans1 zum 2. Mal den Namen erfragen
         #sql [trans1] {
                        SELECT name INTO :phantom FROM angestellte
                        WHERE ang_id = 'A1'
                       };
         System.out.println("[trans1]: Name jetzt '" + phantom + "'");
         // Auf den alten Namen zuruecksetzen
         #sql [trans2] {
                        UPDATE angestellte SET name = 'Fritz'
                        WHERE ang_id = 'A1'
                       };
         // Speichern
         #sql [trans2] { COMMIT };
         // Mal wieder mit trans1 den Namen lesen
         #sql [trans1] {
                        SELECT name INTO :phantom FROM angestellte
                        WHERE ang_id = 'A1'
                       };
          System.out.println("[trans1]: Name jetzt '" + phantom + "'");
         }
```

```
      catch(Exception e) { e.printStackTrace(); }
    }
}
```

2.3.4.2 Dynamisches SQL

Das ist lediglich eine konzeptionelle Erweiterung von Embedded SQL. SQL-Anweisungen werden zur Laufzeit generiert und ausgeführt. Datenbankoperationen müssen deshalb nicht im voraus feststehen.
Die Abfrage muß nicht vom Programmierer entwickelt sein. Die Anweisung wird üblicherweise in einem Puffer erstellt und mit folgendem Befehl ausgeführt:

```
EXEC SQL PREPARE S FROM :order;
```

Hier kann der FETCH-Befehl ohne Angaben von Variablen ausgeführt werden. In einem solchen Fall gibt die Anweisung einen Zeiger auf ein Feld zurück bzw. die Summe der Feldelemente zurück. Das Programm überträgt anschließend die Felddaten in eigene Variablen.
Vier Methoden können für das Einbeziehen dynamischer SQL-Anweisungen herangezogen werden:

1. Methode
Sie akzeptiert eine dynamische SQL-Anweisung, die sofort (über das `EXECUTE IMMEDIATE`) Kommando ausgeführt wird. Die Anweisung darf keine Abfrage (SELECT-Anweisung) sein und darf auch keine Platzhalter für Eingabe-"host"-Variable besitzen
2. Methode
Sie akzeptiert eine dynamische SQL-Anweisung, die über die Kommandos `PREPARE` und `EXECUTE` ausgeführt wird. Die Anweisung darf keine Abfrage sein, die Anzahl der Platzhalter für "Eingabe-host-Variable" und die Datentypen für Eingabe-"host"-Variable müssen zur Precompile-Zeit bekannt sein.
3. Methode
Hier ist die SQL-Anweisung eine dynamische Anfrage, die über das `PREPARE`-Kommando mit den Cursor-Kommandos `DECLARE`, `OPEN`, `FETCH` und `CLOSE` verarbeitet wird. Die Anzahl der ausgewählten Merkmale, die Anzahl der Platzhalter für "Eingabe-host-Variable", die Datentypen für "Eingabe-host-Variable" müssen zur Precompile-Zeit bekannt sein.
4. Methode
Sie akzeptiert oder baut eine dynamische SQL-Anweisung auf, die mit Hilfe von Deskriptoren verarbeitet werden. Ein Deskriptor ist ein Speicherbereich, der für Programm und ORACLE eine vollständige Bearbeitung der Variablen einer dynamischen SQL-Anweisung enthält.

Bsp.: Ein C-Programm mit „dynamischen SQL“

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
VARCHAR uid [20];
VARCHAR pwd [20];
VARCHAR dbstring [20];
VARCHAR ang_id [3];
VARCHAR name[10];
VARCHAR gebjahr[9];
VARCHAR abt_id[2];
VARCHAR job_id[2];
VARCHAR order[255];

EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL INCLUDE sqlca;

main (int argi, char * argv [])
{
   char input[255];

   strcpy (uid.arr, "xyz12345");
   uid.len = strlen(uid.arr);
   strcpy (pwd.arr, "xyz12345");
   pwd.len = strlen(pwd.arr);
   strcpy (dbstring.arr, "rfhs8012_ora8");
   dbstring.len = strlen(dbstring.arr);


   EXEC SQL WHENEVER SQLERROR GOTO error;

   EXEC SQL CONNECT :uid IDENTIFIED BY :pwd USING :dbstring;

   printf ("Connected to ORACLE user : %s \n", uid);

   EXEC SQL WHENEVER NOT FOUND GOTO loop;

   for (;;)
   {
      loop:

      printf ("Please enter your SQL-Command\n");
      strcpy (input,"");
      gets (input);

      printf ("%s\n", input);

      if (strcmp (input, "exit") == 0) exit (0);

      strcpy(order.arr, input);
      order.len = strlen(order.arr);

      EXEC SQL PREPARE S1 FROM :order;
      EXEC SQL DECLARE C1 CURSOR FOR S1;
      EXEC SQL OPEN C1;
      for (;;)
      {
       EXEC SQL FETCH C1
         INTO :ang_id, :name, :abt_id, :job_id;
       printf ("%s %s %s %s\n", ang_id.arr, name.arr, abt_id.arr,
                job_id.arr);
      }
   }

   error:
   printf ("Fehler aufgetreten\n");
   printf ("%d\n", sqlca.sqlcode);
   exit (-1);
}
```

2.3.4.3 Call-Schnittstelle (CLI)

SQL-Anweisungen werden in Funktionsaufrufen als String-Parameter übergeben und zur Laufzeit vom Datenbanksystem interpretiert. Diese spezielle Form des dynamischen SQL wird im Rahmen von SQL3-Standards als SQL/CLI (Call Level Interface) ebenfalls normiert. Der wesentliche Vorteil der Call-Schnittstelle gegenüber Embedded-SQL ist: Realisierung der SQL-Befehle über C-Funktionsaufrufe. Es ist kein Precompiler nötig.

2.3.4.3.1 ODBC

Die wohl am weitesten verbreitete Version einer Call-Schnittstelle ist zur Zeit ODBC unter Microsoft Windows. Im wesentlichen stützt sich ODBC auf einen CLI-Standard von X/Open und der SQL Access Group. Diese Spezifikation wurde bei der ISO als Erweiterung von SQL vorgeschlagen und hat große Chancen als SQL/CLI-Norm übernommen zu werden.

Die ODBC-Ebenen

Eine ODBC-Anwendung hat fünf logische Ebenen (Layer): Anwendung, ODBC-Schnittstelle, Treibermanager, Treiber und Datenquelle.

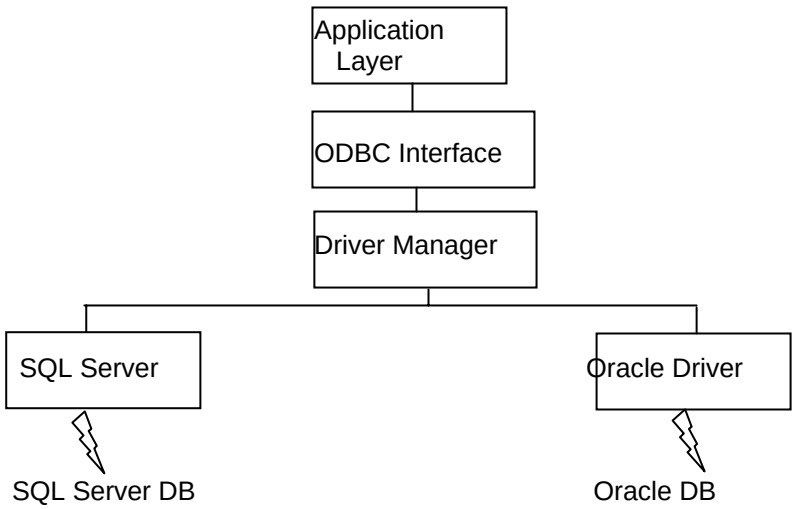


Abb.: Architekturschema mit fünf ODBC-Ebenen

Die Anwendungs-Ebene ist in Sprachen wie Java, Visual Basic und C++ geschrieben. Die Anwendung verwendet die ODBC-Funktionen in der ODBC-Schnittstelle zur Interaktion mit der DB.
Die Treibermanager-Ebene ist Teil von ODBC. Da eine Anwendung nicht mit mehr als einer DB verbunden werden kann, stellt der Treiber-Manager sicher, daß das „richtige DBMS“ alle Programmaufrufe bekommt, die an es gerichtet sind. Der Treiber ist die Komonente, die die spezifische DB kennt. Normalerweise ist der Treiber einer spezifischen DB zugewiesen, z.B. einem Access-Treiber, einem SQL-Server-Treiber, einem Oracle-Treiber. Der Treiber kümmert sich um: „Das Schicken der Anfrage an die DB“, „das Zurückbekommen der Datenbank-Daten“,

„die Übermittlung der Daten an die Anwendung“. Bei Datenbanken, die sich in einem lokalen Netzwerk oder im Internet befinden, übernimmt der Teiber auch die Netzwerkkommunikation.
ODBC definiert für die Treiber Konformitätslevel für SQL- und API-Grammatik.

Typ	Konformitätsstufe	Beschreibung
API-Konformitätsstufe	Core	umfaßt alle Funktionen in der SAG CLI-Spezifikation: - Ausrichten und Freigeben einer Verbindung, eines Statement und von Umgebungs-Handles. - Vorbereiten und Ausführen von SQL-Statements - Einholen von Ergebnisdaten und Informationen -Fähigkeiten zur Ausführung von Transaktionen und zum Aufrollen von Transaktionen von Anfang an
	Stufe 1	ist eine Core API plus der Fähigkeit - Einholen und Versenden partieller Datensätze - Holen von Katalog-Informationen - Verschaffen von Informationen über Treiber und Datenbankfähigkeiten
	Stufe 2	umfaßt Core API plus Stufe 1 plus der Fähigkeit - Verarbeiten von Arrays als Parameter - Verarbeiten scrollbarer Cursors - Aufrufe der Übersetzungs-DLL - ...
SQL-Grammatik-konformitätsstufe	Minimale Grammatik	Erzeugen von Table- und „drop“ Table-Funktionen in der DLL, die Funktion Auswahl, Einfügen, Aktualisieren, Löschen in der DML, einfache Ausdrücke.
	Core Grammar	Minimale Grammatik plus ALTER TABLE, Erzeugen und Aufheben eines Index bzw. eines View für DDL, volle SELECT-Fähigkeiten für DML
	Extended Grammar	Fügt Funktionalitäten wie Outer Join, positioned update, delete, weitere Ausdrücke , weitere Datentypen, Prozeduraufrufe, usw. hinzu

Abb.: API- und SQL-Konformitäts-Level

ODBC-Funktionen und Befehlssatz

Alle ODBC-Funktionen haben das „Präfix“ SQL und können einen oder mehrere Parameter vom Typ Input (für den Treiber) oder Output (von dem Treiber) umfassen:

Bestimmen der Umgebungs- und Verbindungs-Handles

Verbinden mit der Datenbank, Ausführen der SQL-Anweisungen, Schließen der Verbindung

Zurücknahme der Handles, Schließen der Datenbank

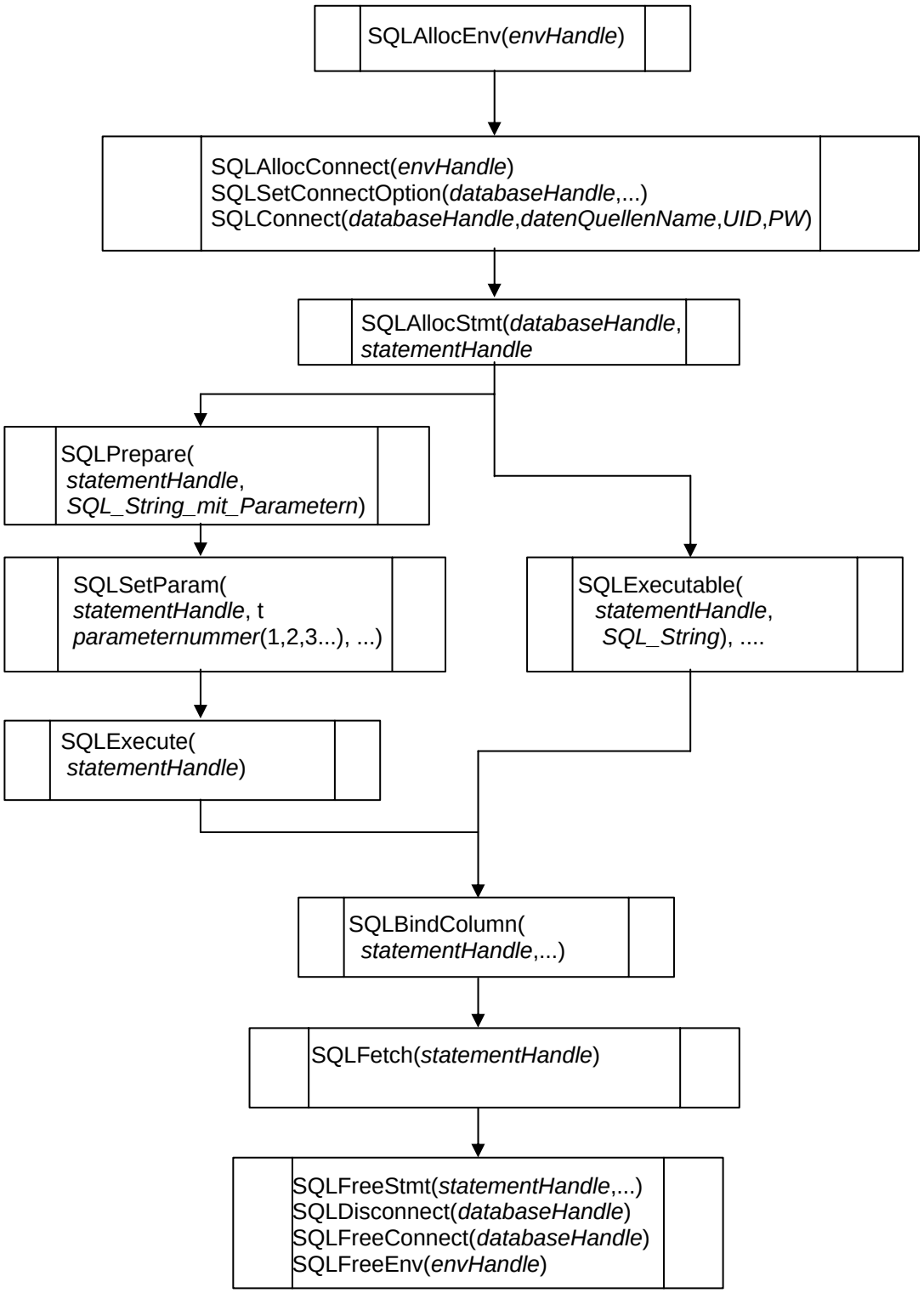


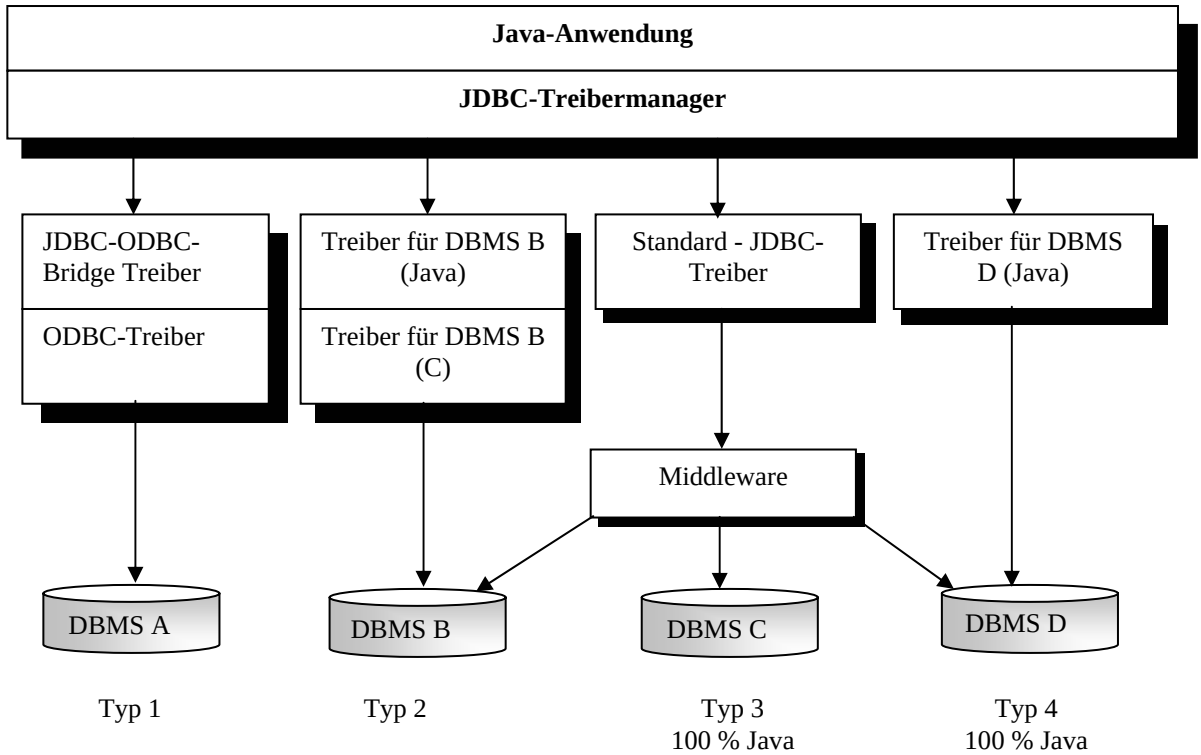
Abb.: Das ODBC-Programmflußschema für ein typisches Programm

2.3.4.3.2 JDBC

Überblick zu JDBC

JDBC steht für Java Database Connectivity. Die JDBC-APIs sind Teil der Enterprise APIs, die von JavaSoft spezifiziert wurden und somit Bestandteil der Java Virtual Machine (JVM) sind. JDBC-Designer haben die API auf das X/Open SQL-Call Level Interface (CLI) aufgebaut. Auch ODBC basiert auf X/Open SQL-CLI. JDBC definiert API-Objekte und Methoden zur Kommunikation mit einer Datenbank. Ein Java-Programm baut zunächst eine Verbindung zur Datenbank auf, erstellt ein Statement-Objekt, gibt SQL-Statements an das zugrundeliegende Datenbankverwaltungssystem (DBMS) über das Statement-Objekt weiter und holt sich Ergebnisse und auch Informationen über die Resultat-Datensätze. JDBC-Dateien und Java-Anwendung / Applet bleiben beim Client, können aber auch vom Netzwerk heruntergeladen werden. Das DBMS und die Datenquellen liegen auf einem Remote Server. Die JDBC-Klassen befinden sich im `java.sql`-Paket. Alle Java-Programme verwenden zum Lesen und Schreiben von Datenquellen Objekte und Methoden des `java.sql`-Pakets.
Ein Programm, das JDBC verwendet, benötigt einen Treiber für die Datenquelle. Es ist die Schnittstelle für das Programm. JDBC besitzt den Driver Manager zur Verwaltung der Treiber und zum Erstellen einer Liste der in den Anwendungsprogrammen geladenen Treiber. JDBC-ODBC-Bridge ist als `jdbcOdbc.class` implementiert und eine native Bibliothek für den Zugriff auf den ODBC-Treiber. Zuerst bildet dieser Treiber JDBC-Methoden auf ODBC-Aufrufe und tritt somit mit jedem verfügbaren ODBC-Treiber in Interaktion.

JDBC-Treibertypen



1. Typ 1 ist eine schnelle Lösung für jede beliebige ODBC-Datenbank. Die JDBC-ODBC-Bridge stützt sich nach unten hin auf einen bereits vorhandenen ODBC-Treiber und greift dabei einfach auf dessen Funktionalität zurück. Dadurch kann man bereits zu einem frühen Zeitpunkt auf jede x-beliebige ODBC-Datenbank zugreifen. Allerdings hat diese Methode einen gewichtigen Nachteil: der JDBC Treiber besteht nicht aus reinem Java-Bytecode, sondern zu einem Großteil auch aus dem ODBC-Treiber, d.h. die Portabilität des Treibers ist nicht gewährleistet. Clients, die diesen Treiber verwenden sind dadurch an die Plattform gebunden, auf der der ODBC-Treiber läuft und sind zusätzlich auf die Installation des ODBC-Treibers auf jedem Client-Rechner angewiesen. Ein Typ 1 Treiber ist daher nur eine Übergangslösung für ODBC-Datenbanken, für die noch kein JDBC-Treiber existiert.
2. Ein Typ 2 Treiber ist eine attraktive Lösung für diejenigen Datenbankhersteller, die noch nicht in der Lage waren, einen eigenen JDBC-Treiber Typ 3 oder Typ 4 zur Verfügung zu stellen. Dieser Treibertyp ist als aufgesetzte Schicht schnell und einfach zu implementieren, da die Grundfunktionalität bereits in einem alten, z.B. in C geschriebenen Treiber, vorhanden ist. Auch hier handelt es sich demnach nur um eine plattformspezifische Übergangslösung.
3. Typ 3 ist ein vollständig in Java geschriebener Treiber und bietet somit die größtmögliche Flexibilität. Er fügt sich nahtlos in das Dreischichtenmodell ein. Er kommuniziert über das Netzwerk mit einer Middleware. Diese bearbeitet sämtliche Anfragen. Ein Beispiel für einen Typ 3 Treiber ist der Treiber der Firma OpenLink. Dieser Treiber kann jede Datenbank ansprechen, die die Middleware unterstützt.
4. Der vierte Typ von JDBC-Treibern ist ebenfalls rein in Java implementiert. Er unterstützt allerdings nicht das Dreischichten- sonder nur das Zweischichtenmodell. Auch der in der Beispielanwendung verwendete JDBC-Treiber von Oracle ist in diese Kategorie einzuordnen. Diese Treiber werden von den Herstellern für ihr DBMS selbst entwickelt und ersetzen nach und nach immer mehr die althergebrachten Typ 2 Treiber, die als Übergangslösungen am Markt existierten.
   „Während ein Typ 4-Treiber aus der Sicht des Anwendungsentwicklers als ein monolithisches Stück Java-Software erscheint, ist die interne Struktur komplexer. Schließlich kann der Treiber in seiner Eigenschaft als Java-Klasse nicht unmittelbar Methoden des Datenbanksystems aufrufen – bislang ist kein relationales DBMS in Java implementiert.“[174]
   Auch hier liegt zwischen Treiber und Datenbank eine Middleware, nur mit dem Unterschied, daß es sich um eine proprietäre Schicht des jeweiligen Herstellers handelt, die nach außen hin unsichtbar und dem Entwickler nicht zugänglich ist.

Grundlagen der JDBC-Anwendung

Der Grundaufbau einer Datenbankanwendung unter Verwendung eines JDBC-Treibers ist immer gleich:

Laden eines JDBC-Treibers

Das Laden der JDBC-Klassen erfolgt über die Import-Anweisung:

```
import java.sql.*;
```

Der Import stellt die JDBC-Klassen zur Verfügung.
Danach muß der Treiber selbst geladen werden. Dies geschieht über die Anweisung:

```
Class.forName( "oracle.jdbc.driver.OracleDriver" );
```

Der Treiber selbst ist eine gewöhnliche Java-Klasse. Das ungewöhnliche an dieser Vorgehensweise ist, daß diese erst zur Laufzeit des Programms hinzugebunden wird.

[174] Rainer Klute: JDBC in der Praxis, Addison-Wesley, S. 41

Hierfür bedient sich die Java Virtual Machine des Classloaders. Die Klasse, die geladen werden soll, muß sich in einem Pfad, einer ZIP- oder JAR- Datei befinden, die im CLASSPATH eingetragen ist[175]. Ist der Classloader nicht in der Lage, dieTreiberklasse hinzuzubinden, erzeugt er eine java.lang.ClassNotFoundException.
Wird der Treiber erfolgreich geladen, meldet er sich mit seinem Initialisierungs-code beim JDBC-Treibermanager an.
Selbstverständlich kann ein Programm viele verschiedene JDBC-Treiber gleichzeitig laden, um verschiedene Datenbanken parallel anzusprechen. Dabei sind die einzelnen Treiber meistens herstellerspezifisch.
Im Falle des Oracle-Treibers wird empfohlen, den JDBC-OCI Treiber zu registrieren. Dies erfolgt mit Hilfe des Aufrufs von

```
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
```

Öffnen einer Datenbank

Das Öffnen der Datenbank kann unter Verwendung zweier Methoden erfolgen. Die erste Methode, die nur für den direkten Aufruf des OCI –Connects geeignet ist, verwendet den Datenbankalias oder TNSName, der in der tnsnames.ora spezifiert wird, wie bereits im vorangegangenen Kapitel über die Konfiguration des SQL*Nets beschrieben wurde.

```
Connection conn =
DriverManager.getConnection ("jdbc:oracle:oci7:@mydatabase","scott",
"tiger");
```

Scott ist in diesem Fall der Username, Tiger das Passwort und @mydatabase der TNSName der Datenbank. Weiterhin ist angegeben, daß es sich um einen Aufruf des OCI7 Treibers handelt.

Eine weitere Möglichkeit, das Öffnen der Datenbank durchzuführen, falls keine sauber erstellte Datei `tnsnames.ora` vorliegt, ist die Ergänzung des Strings um eine gültige SQL*Net Namensspezifikation:

```
Connection conn =
DriverManager.getConnection
      ("jdbc:oracle:oci7:@(description=(address=(host=myhost)
      (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))","scott",
      "tiger");
```

Man erkennt die Ähnlichkeit zu den Eintragungen in der Datei `tnsnames.ora`. Im Connectstring werden direkt der Hostname, das Protokoll und der Port angegeben. Weiterhin enthält er eine SID (System Identifier), hier den String ‚`orcl`‘. Username und Passwort bleiben gleich. Diese Aufrufe gelten aber nur für eine Verbindung über das Oracle Call Interface von einem normalen Java-Programm aus.
Im Falle eines Applets wird der Zugriff auf die Datenbank über das Internet gewünscht. Selbstverständlich ist für diese Art der Verbindung das SQL*Net nutzlos, da nicht jeder Client-Rechner über eine Oracle-Installation verfügen muß, um online auf eine Datenbank zuzugreifen.
Für diese spezielle Zugriffsmöglichkeit wurde der Oracle JDBC Thin Treiber konzipiert.

[175] vgl. Installationshinweise

```
Connection conn =
DriverManager.getConnection
      ("jdbc:oracle:thin:scott/tiger@irgendwo.nirwana.de:1527:oradb" );
```

Im Connectstring wird das ‚OCI7' durch ‚thin' ersetzt. Name und Passwort bleiben gleich, die Änderung in der Zusammensetzung des Strings ist optional.
Nach dem @ wird der komplette Hostname eingetragen. Der Port 1527 ist derjenige, auf dem die Datenbank mit dem Identifizierer `oradb` über ihre Listener auf Anfragen wartet. Das verwendete Protokoll ist selbstverständlich TCP/IP.
Auch für diesen String gibt es eine zweite Möglichkeit:

```
Connection conn =
DriverManager.getConnection
      ("jdbc:oracle:thin:@(description=(address=(host=myhost)(protocol=tc
      p)
            (port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Der Connectstring im Falle eines JDBC-Aufrufs nennt sich JDBC-URL. Die Methode `getConnection()` gehört zur Klasse DriverManager. „Der Treibermanager überprüft, welcher der geladenen Treiber den angegebenen JDBC-URL öffnen kann, und ruft die entsprechende Methode dieses Treibers auf."[176]
Als Ergebnis liefert die Methode ein Objet vom Typ Connection. Dieses Connection Objekt ist für alle weiteren Datenbankoperationen notwendig.

Senden von SQL-Anweisungen an die Datenbank

Der Client sendet seine SQL-Anweisungen über ein Objekt vom Typ Statement oder PreparedStatement an die Datenbank. Um dieses Objekt zu generieren, wird eine Methode der Klasse Connection aufgerufen. Im nachfolgenden Beispiel sei die Variable con eine Instanz der Klasse Connection:

```
Statement stmt = con.createStatement();
```

Die eigentliche SQL-Anweisung, INSERT, DELETE oder UPDATE, werden in einer Stringvariablen definiert, z.B.

```
String SQL = "DELETE * FROM …";
```

Der oben definierte String wird über das `stmt`-Objekt vom Typ Statement an die Datenbank geschickt:

```
ResultSet rset = stmt.executeUpdate( SQL );
```

Die Methode `executeUpdate()` liefert 0 oder die Anzahl der betroffenen Datensätze zurück. Sie wird für INSERT, DELETE und UPDATE verwendet, sowie für DDL-Befehle, die keine Ergebnismenge erzeugen.
Eine SELECT-Anweisung wird über die Methode:

```
ResultSet rset = stmt.executeQuery( SQL );
```

auf der Datenbank abgesetzt. Dabei wird eine Ergebnismenge im ResultSet zur Verfügung gestellt. Die Bearbeitung solcher ResultSets soll später erörtert werden.

[176] Rainer Klute: JDBC in der Praxis, Addison-Wesley, S. 74

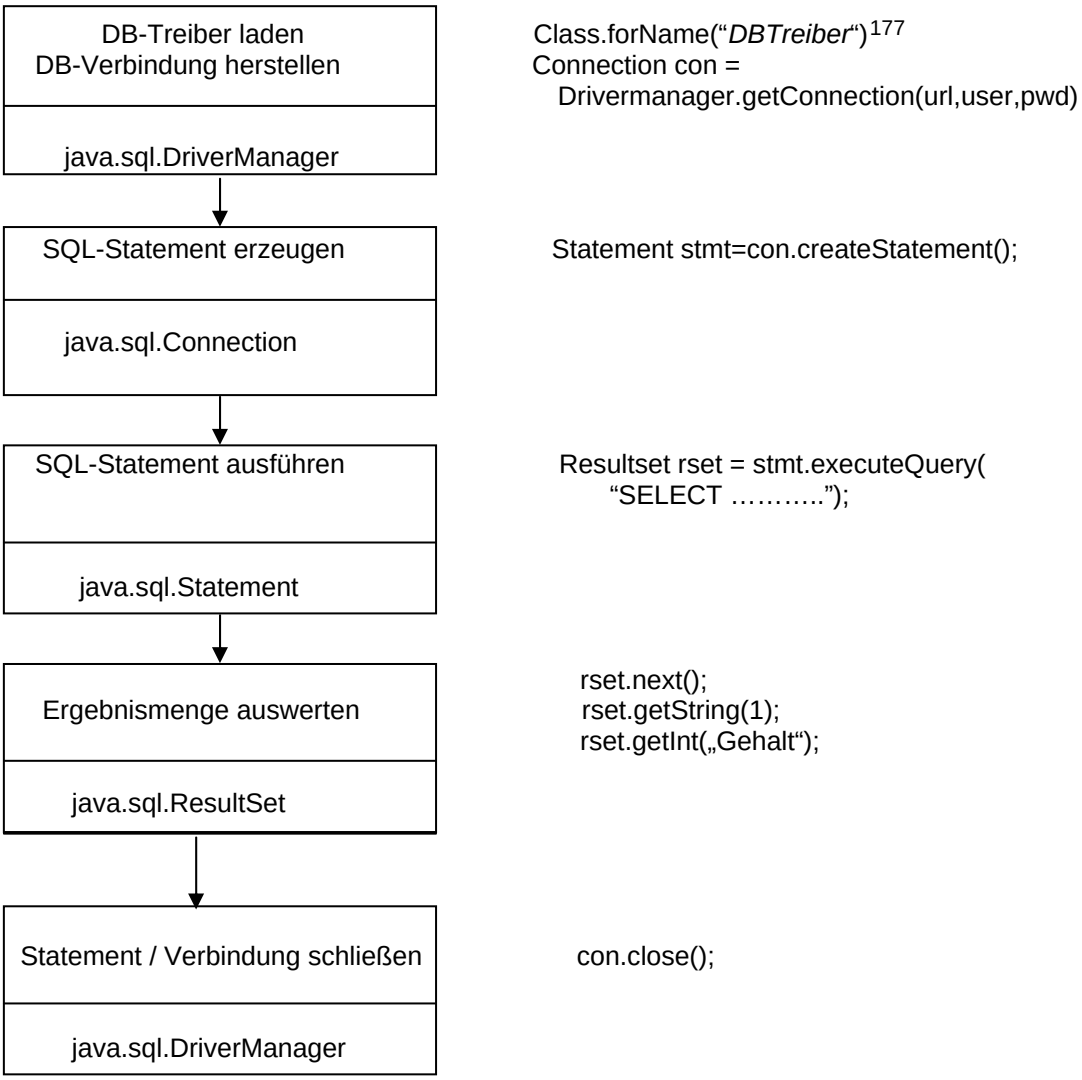


Abb.: Prinzip einer JDBC-Anwendung

Statement-Arten

177 Laden einer Klasse ohne ein Instanz zu erzeugen. Referenzieren einer Klasse, deren Name noch nicht bekannt ist. Da eine ClassNotFoundexception ausgelöst werden könnte, sollte die Anweisung in einem try / catch – Block stehen.

Klassen und Interfaces der JDBC-Schnittstelle

JDBC ist als `java.sql`-Paket implementiert. Dieses Paket enthält alle JDBC-Klassen und Methoden. Klassen im `java.sql`-Paket sind:

`java.sql.Driver`, `java.sql.DriverManager`, `java.sql..PropertyInfo`,
`java.sql.Connection`, `java.sql.Statement`, `java.sql.PrepareStatement`,
`java.sql.CallableStatement`, `java.sql.ResultSet`, `java.sql.SQLException`,
`java.sql.SQLWarning`, `java.sql.Database.MetaData`,
`java.sql.ResultSetMetaData`,
`java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`,
`java.sql.Numeric`, `java.sql.Types`, `java.sql.DataTruncation`.

Die Klassen `DriverManager` und `Driver` und verwandte Methoden

Die höchste Klasse in der Hierarchie ist der `DriverManager` mit Treiberinformationen, Statusinformationen. Wenn ein Treiber geladen ist, hat er sich beim `DriverManager` registriert.
Der DriverManager ist in der Lage, eine größere Anzahl von JDBC-Treibern zu verwalten. Sollte in der Systemumgebung ein JDBC-Treiber als Standard definiert sein, so versucht der DriverManager, diesen zu laden. Zudem können dynamisch jederzeit weitere Treiber nachgeladen werden. Dies geschieht ber den Aufruf des ClassLoaders über `Class.forName()`.
Anhand der JDBC-URL versucht der DriverManager bei Aufruf der Methode `getConnection()` den richtigen Treiber ausfindig zu machen und die Datenbankverbindung zu initialisieren.

Methoden der Klasse `jdbc.sql.Driver`:

Methoden-Name	Parameter	Return-Typ
connect	(String *url*,java.util.Properties *info*)	Connection
acceptsURL	(String *url*)	boolean
getPropertyInfo	(String url,java.util.Properties *info*)	DriverPropertyInfo[]
getMajorVersion	()	int
getMinorVersion	()	int
jdbcCompliant	()	boolean

Methoden der Klasse `java.sql.DriverManager`:

Methode	Parameter	Return-Type
getConnection	(String *url*,java.util.Properties *info*)	Connection
getConnection	(String *url*,String *user*, String *password*)	Connection
getConnection	(String *url*)	Connection
getDriver	(String *url*)	Driver
registerDriver	(java.sql.Driver *driver*)	void
deregisterDriver	(Driver *driver*)	void
getDrivers	()	Java.util.Enumeration
setLoginTimeout	(int *seconds*)	void
getLoginTimeout	()	int
setLogStream	(java.io.Printstream *out*)	void
getLogStream	()	Java.io.PrintStream
println	(String *message*)	void

Klassen-Initialisierungsroutinen

Methode	Parameter	Return-Type
intialize	()	void

Die Klasse `Connection`

Eine Connection repräsentiert immer eine Sitzung mit einer spezifischen Datenbank. Jede Connection definiert einen Kontext. Im Gültigkeitsbereich dieses Kontexts werden SQL-Anweisungen an eine Datenbank gesandt und gegebenenfalls Ergebnismengen zurückgeliefert.
Weiterhin können über eine Connection Metadaten über die Datenbank erhalten werden.
Jede Connection ist per Default in den AutoCommit Modus geschaltet, d.h., daß jede Änderung des Datenbestands sofort dauerhaft gültig wird.
Die Attribute einer Connection beziehen sich ausnahmslos auf die Transaction-Isolation.

`java.sql.Connection`-Methoden und Konstanten

Methode	Parameter	Return-Type
createStatement	()	Statement
prepareStatement	(String sql)	PreparedStatement
prepareCall	(String sql)	CallableStatement
nativeCall	(String sql)	String
close	()	void
isClosed	()	boolean
getmetaData	()	DatabaseMetaData
setReadOnly	(boolean readOnly)	void
isReaddOnly	()	boolean
setCatalog	(String catalog)	void
getCatalog	()	String
setAutoClose	(boolean autoClose)	void
getAutoClose	()	boolean
getWarnings	()	SQLWarning
setAutoCommit	(boolean autoCommit)	void
getAutoCommit	()	boolean
commit	()	void
rollback	()	void
setTransactionIsolation	(int level)	void
getTransactionIsolation	()	int

Die Statement-Interfaces

In JDBC gibt es drei Typen von Statement-Objekten zur Interaktion mit SQL: Statement, PreparedStatement, CallableStatement.

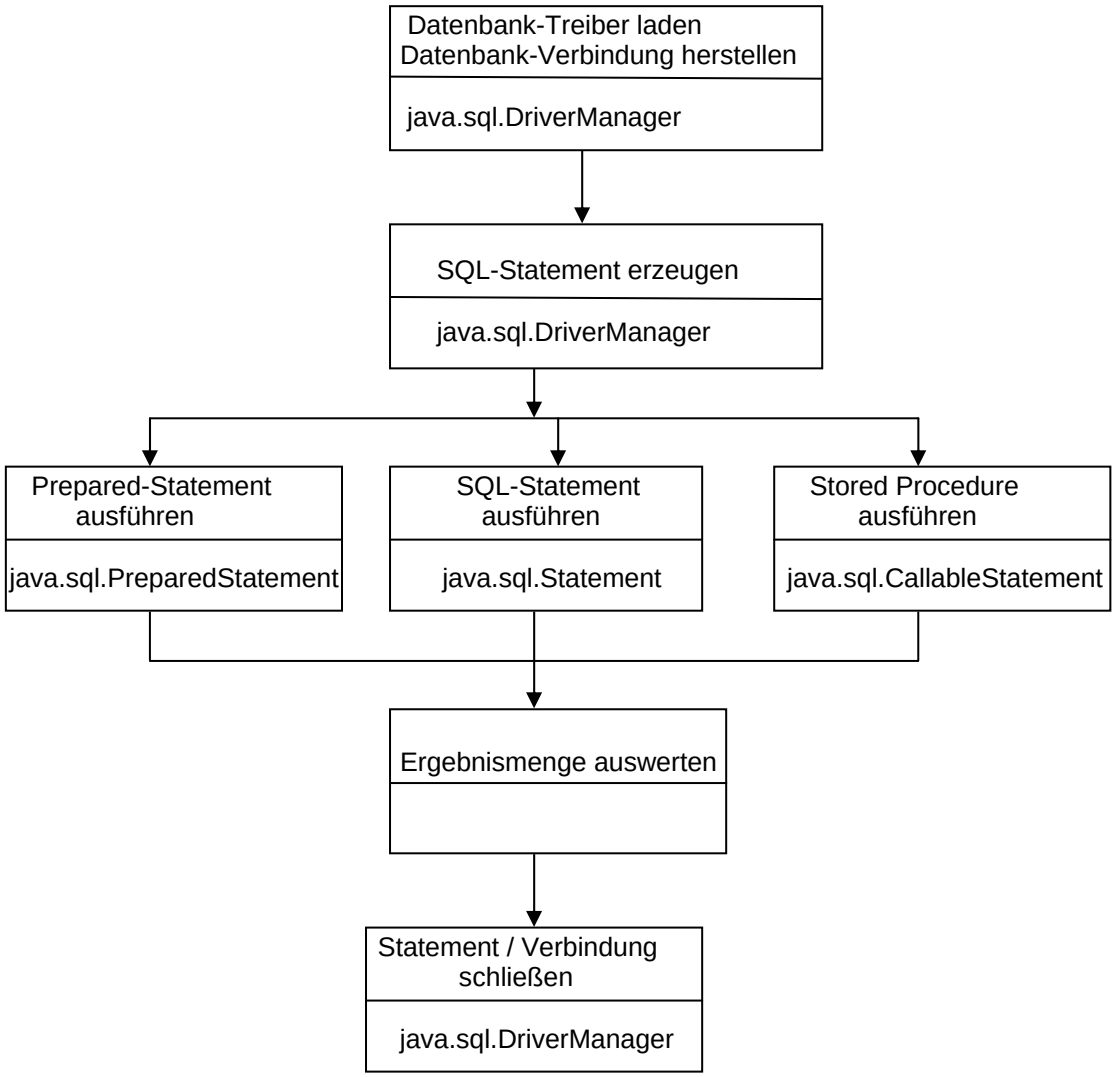


Abb.: Statement-Arten

Das Interface `Statement`

Ein Statement-Objekt wird mit createStatement() des Connection-Objekts erzeugt.

Methodenname	Parameter	Rückgabetyp
executeQuery	(String *sql)*	ResultSet
executeUpdate	(String *sql)*	int
execute	(String *sql)*	boolean
getMoreResults	()	boolean
close	()	void
getMaxFieldSize	()	int
setMaxFieldSize	(int max)	void
getMaxRows	()	int
setMaxRows	(int max)	void
setEscapeProcessing	(boolean enable)	void
getQueryTimeout	()	int
setQueryTimeout	(int seconds)	void
cancel	()	void
getWarnings	()	Java.sql.SQLWarning
clearWarnings	()	void
setCursorName	(String name)	void

getResultSet	()	ResultSet
getUpdateCount	()	int

Die wichtigsten Methoden sind `executeQuery`, `executeUpdate()` und `execute()`. Falls ein Statement-Objekt mit einem SQL-Statement erzeugt wird, dann nimmt `executeQuery()` einen SQL-String an. Sie gibt die SQL-Zeichenkette über den Treibermanager an die zugrundeliegende Datenquelle weiter und bekommt das `ResultSet` für das Anwendungsprogramm. `executeQuery()` gibt nur ein `ResultSet` zurück. In den Fällen, in denen mehr als ein `ResultSet` erhalten wird, sollte `execute()` verwendet werden.
Für SQL-Anweisungen, die kein `ResultSet` zurückgeben (`update`, `delete`, DDL-Statements), gibt es `executeUpdate()`. Diese Methode nimmt einen SQL-String und gibt einen `Integer` (Zahl der Spalten, die vom SQL-Statement betroffen sind,) zurück.

Das Interface `PreparedStatement`

Beim PreparedStatement-Objekt bereitet das Anwendungsprogramm ein SQL-Statement mit der java.sql.Connection.prepeareStatement()-Methode vor. Eine PreparedSatetement-Methode nimmt eine SQL-Zeichenkette, die an das zugrundeliegende DBMS weitergegeben wird. Das DBMS führt die SQL-Anweisung aber nicht aus. Die Methoden executeQuery(), executeUpdate() und execute() nehmen keine Parameter auf, sondern nur Aufrufe für das DBMS, das (bereits optimierte) SQL-Statement auszuführen.

Das Interface `CallableStatement`

Ein `CallableStatement`-Objekt wird von der prepared-Methode eines Connection-Objekts erzeugt.

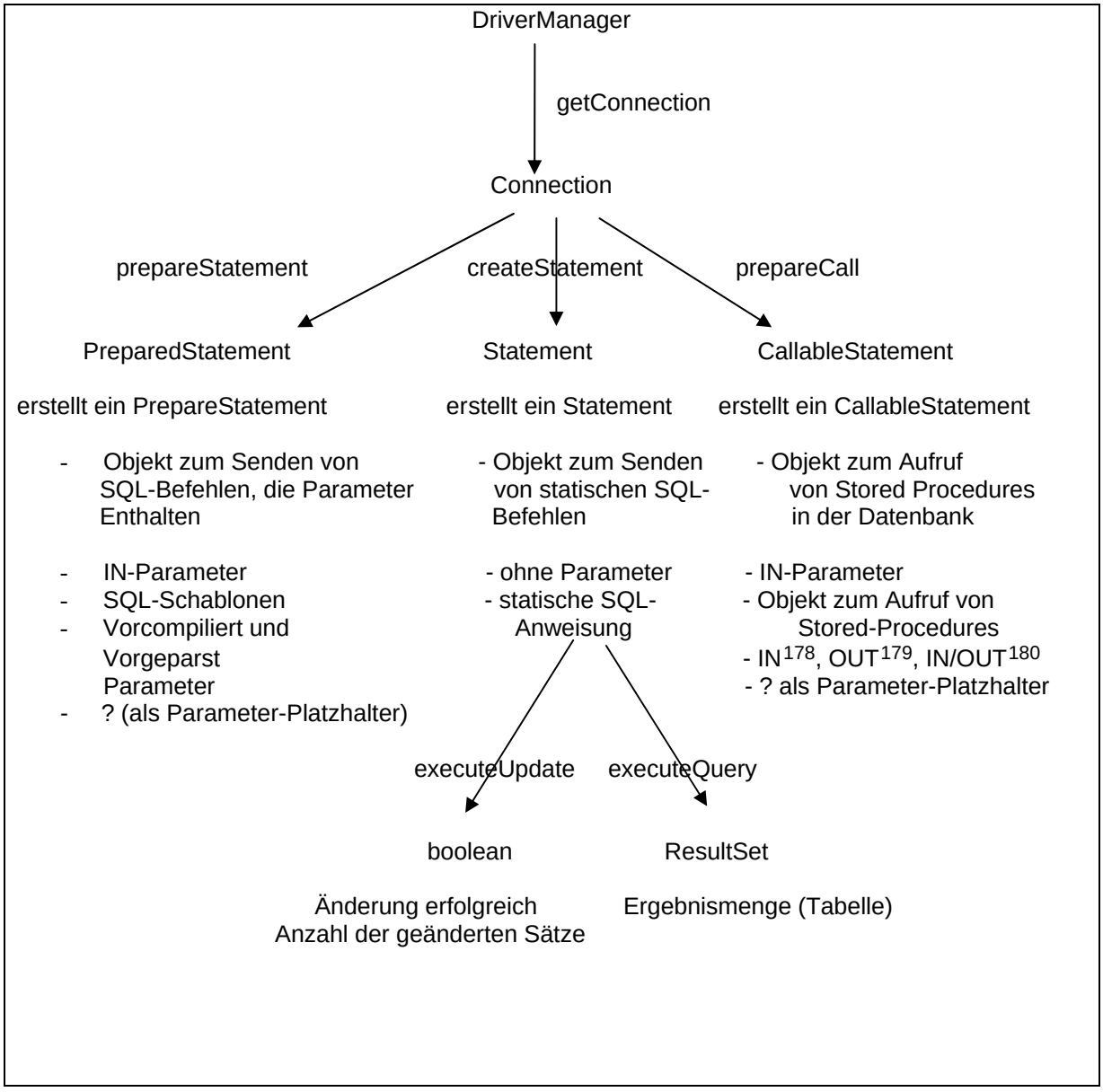


Abb.:

Die Klasse `ResultSet`

Die Ausführung eines Statements generiert eine Art virtuelle Tabelle, die in einem ResultSet abgespeichert ist. Die Datensätze sind sequentiell angeordnet. Innerhalb eines Datensatzes kann in beliebiger Reihenfolge auf ein Attribut positioniert werden.
Ein ResultSet stellt einen Cursor zur Verfügung, der auf den jeweils aktuellen Datensatz verweist. Nach der Initialisierung des ResultSet steht dieser Cursor allerdings vor dem ersten Datensatz. Mit der Methode `next()` kann zum ersten Datensatz gesprungen werden.

[178] IN-Parameter müssen gesetzt sein, Übergabe von Werten
[179] OUT-Parameter enthalten Rückgabeparametere
[180] IN/OUT-Parameter: Übergabeparameter, die ihren Wert verändern können

Die Spalten in einem Datensatz können über den Index, beginnend bei 1, oder aber den Spaltennamen angesprochen werden. Der Spaltenname wird case insensitive behandelt. Das Auslesen der Werte erfolgt über eine Vielzahl von `getXXX()`-Methoden, entsprechend dem zu lesenden Datentypen. Diese Methoden führen eine Konvertierung des datenbankspezifischen Datentyps in den zugehörigen Java-Typen durch.
Ein ResultSet wird vom ausführenden Statement geschlossen sobald das Statement geschlossen wird, wenn es neu ausgeführt wird, oder aber wenn zur nächsten Ergebnismenge gesprungen wird, für den Fall daß mehrere Ergebnismengen vorliegen.
Der Index, die Typen und Eigenschaften der Spalten eines ResultSet können durch das ResultSetMetaData-Objekt abgefragt werden, das von der `getMetaData()`-Methode bereitgestellt wird.

`java.sql.ResultSet-Methoden`

Methodenname	Parameter	Rückgabetyp
next	()	boolean
close	()	void
wasNull	()	boolean

Beschaffen der Datenwerte über die Position

Methodenname	Parameter	Rückgabetyp
getAsciiStream	(int *columnIndex*)	java.io.Input.Stream
getBinaryStream	(int *columnIndex*)	Java.io.Input.Stream
getBoolean	(int *columnIndex*)	boolean
getByte	(int *columnIndex*)	byte
getBytes	(int *columnIndex*)	byte[]
getDate	(int *columnIndex*)	Java.sql.Date
getDouble	(int *columnIndex*)	double
getFloat	(int *columnIndex*)	float
getInt	(int *columnIndex*)	int
getLong	(int *columnIndex*)	long
getNumeric	(String *columnIndex*,int *scale*)	Java.sql.Numeric
getObject	(String *ColumnIndex*)	Object
getShort	(int *columnIndex*)	short
getString	(int *columnIndex*)	String
getTime	(int *columnIndex*)	Java.sql.Time
getTimeStamp	(int *columnIndex*)	Java.sql.Timestamp
getUnicodeStream	(int *columnIndex*)	java.io.InputStream

Beschaffen der Datenwerte über den Spalten-Name

Methodenname	Parameter	Rückgabetyp
getAsciiStream	(String *columnName*)	java.io.Input.Stream
getBinaryStream	(String *columnName*)	Java.io.Input.Stream
getBoolean	(String *columnName*)	boolean
getByte	(String *columnName*)	byte
getBytes	(String *columnName*)	byte[]
getDate	(String *columnName*)	Java.sql.Date
getDouble	(String *columnName*)	double
getFloat	(String *columnName*)	float
getInt	(String *columnName*)	int
getLong	(String *columnName*)	long
getNumeric	(String *columnName*,int *scale*)	Java.sql.Numeric
getObject	(String *ColumnName*)	Object

getShort	(String *columnName*)	short
getString	(String *columnName*)	String
getTime	(String *columnName*)	Java.sql.Time
getTimeStamp	(String *columnName*)	Java.sql.Timestamp
findColumn	(String *ColumnName*)	int
getWarnings	()	SQLWarning
clearWarnings	()	void
getCursorName	()	String
getMetaData	()	ResultSetMetaData

`getMetaData()` gibt die Metainformationen über ein `ResultSet` zurück. Die Methoden der Klasse `DatabaseMetaData` geben ebenfalls die Ergebnisse in der ResultSet-Form zurück. Mit den Method `getWarnings()` können Warnungen überprüft werden.

Das Interface `DatabaseMetaData`

Ein `DatabaseMetaData`-Objekt und seine Methoden (über 100) übermitteln Informationen zur zugrundeliegenden Datenbank.

Methodenname	Parameter	Rückgabetyp
allProceduresAreCallable	()	boolean
allTablesAreSelectable	()	boolean
getURL	()	String
getUserName	()	String
isReadOnly	()	boolean
nullsAreSortedHigh	()	boolean
nullsAreSortedLow	()	boolean
nullsAreSortedAtStart	()	boolean
nullsAreSortedAtEnd	()	boolean
.....	...	...

Das Interface `ResultSetMetaData`

Ein `ResultMetaData`-Objekt kann verwendet werden, um mehr über Typ und Eigenschaften von Tabellenspalten in einem ResultSet herauszufinden. Mit den Methoden `getColumnLabel()` und `getColumnDisplaySize()` eines `ResultSetMetaData`-Objekts entstehen Programme, die ResultSets genersich bearbeiten.

Methodenname	Parameter	Rückgabetyp
getColumnCount()	()	boolean
isAutoIncrement	(int *column*)	boolean
isCaseSensitive	(int *column*)	boolean
isSearchable	(int *column*)	boolean
isCurrency	(int *column*)	boolean
isNullable	(int *column*)	int
isSigned	(int *column*)	boolean
getColumnDisplaySize	(int *column*)	int
getColumnLabel	(int *column*)	String
getColumnName	(int *column*)	String
getSchemaName	(int *column*)	String
getPrecision	(int *column*)	int
getScale	(int *column*)	int
getTableName	(int *column*)	String
getCatalogName	(int *column*)	String

getColumnName	(int *column*)	String
getColumnType	(int *column*)	int
getColumnTypeName	(int *column*)	String
isReadOnly	(int *column*)	boolean
isWritable	(int *column*)	boolean
isDefinitelyWritable	(String columnName)	boolean

Die Klasse `SQLException`

Sie umfaßt Fehlermeldungen beim Datenbankzugriff

Methodenname	Parameter	Rückgabetyp
SQLException	(String *reason*,String *SQLState*,int *vendorCode*)	SQLException
SQLException	(String *reason*,String *SQLState*)	SQLException
SQLException	(String *reason*)	SQLException
SQLException	()	SQLException
getSQLState	()	String
getErrorCode	()	int
getNextException		SQLException
setNextException	SQLException e)	void

Die Klasse `SQLWarning`

„Warnings“ werden an das Objekt, das die `Warning` verurscht, angehänt. Die Überprüfung auf Warnungen erfolgt mit der Methode `getWarning()`, die für alle Objekte verfügbar ist.

Methodenname	Parameter	Rückgabetyp
SQLWarning	(String *reason*, String *SQLState*,int *vendorCode*)	SQLWarning
SQLWarning	(String reason,String *SQLState*)	SQLWarning
SQLWarning	(String *reason*)	SQLWarning
SQLWarning	()	SQLWarning
getNextWarning	()	SQLWarning
SetNextWarning	(SQLWarning *w*)	void

Hinweise zur Fehlerbehandlung

Jede Funktion, die eine Operation in Verbindung mit der Datenbank durchführen soll, muß entweder eine

```
throws SQLException
```

Anweisung beinhalten, oder den Anweisungsteil in einem `try{}`-Block durchführen.
Die Beispielanwendung verwendet in jedem Fall die `try{}`-Kapselung, um auf einzeln auftretende Fehler individuell reagieren zu können.
Generell existiert für jeden möglichen SQL-Fehler (Exception) eine eigene Fehler-nummer und -meldung. Die standardisierte Fehlernumerierung erfolgt üblicherweise durch eine Variable SQLSTATE. Im Fall des JDBC-Treibers von Oracle scheint das RDBMS diesen Standard jedoch nicht zu unterstützen (die entsprechende Funktion für das Auslesen des SQLSTATES existiert zwar im SQL-Package von Java, wird aber nicht verwendet).

Oracle bietet produktspezifisch eine Ausgabe des sogenannten SQLCODE. Leider wird dadurch die Portabilität eines JDBC-Clients, der unter Oracle entwickelt wurde, vermindert.
Die entsprechende Funktion zur Abfrage des SQLCODE ist eine Methode von SQLException und lautet `getErrorCode()`.
Während SQLSTATE gewöhnlich durch eine CHAR Repräsentation der Fehlernummer
dargestellt wird, liefert der SQLCODE einen Integer Wert.

Weitere JDBC-Klassen

Die Klasse java.sql.Date

Die Klasse java.sql.Time

Die Klasse java.sql.Types

java.sql.Types-Konstanten:

Die Klasse java.sql.Numeric:

Transaktionsbetrieb

Normalerweise wird jedes SQL-Statement, das via JDBC an die Datenbank übermittelt wird, sofort und dauerhaft auf dem Datenbestand durchgeführt, das heißt, es wird ein automatischer COMMIT abgesetzt.
In manchen Situationen ist es für den Erhalt der Datenkonsistenz zwingend erforderlich, daß mehrere Aktionen erfolgreich abgeschlossen sein müssen, bevor die Änderungen wirksam werden können, d.h. entweder alle SQL-Anweisungen müssen erfolgreich ausgeführt werden, oder aber keine!
Die Anwendung muß, um in den Transaktionsbetrieb zu schalten, die AutoCommit Funktion abschalten, die beim Verbindungsaufbau standardmäßig aktiviert wird. Da es sich um eine Eigenschaft der Verbindung zur Datenbank handelt, wird hierfür eine Methode der Klasse Connection verwendet:

```
Connection.setAutoCommit(false);
```

Um nun zu einem definierten Zeitpunkt die Änderungen wirksam werden zu lassen oder sie gänzlich zu verwerfen, kann auf der Datenbank mit `Connection.commit()` jegliche Änderung bestätigt werden, oder mit `Connection.rollback()` der Urzustand des Datenbestandes wieder hergestellt werden. Ob nun AutoCommit ein- oder ausgeschaltet ist, läßt sich über

```
Connection.getAutoCommit()
```

feststellen.
Die sogenannte Transaktions-Isolation regelt, wie Transaktionen voneinander abge-schottet sind. JDBC definiert fünf Abstufungen, die sich über `Connection.setTransactionIsolation()` einstellen lassen. Die aktuelle

Einstellung verrät ein Aufruf von `Connection.getTransactionIsolation()`.[181]
Die einzelnen Stufen der Isolation sind als Konstanten in der Connection Klasse definiert.

TRANSACTION_NONE	Keine Unterstützung von Transaktionen
TRANSACTION_READ_UNCOMMITTED	Transaktion B kann Daten lesen, die Transaktion A geändert, aber noch nicht per commit() dauerhaft an die Datenbank übergeben hat („Dirty Read“).
TRANSACTION_READ_COMMITTED	„Dirty Reads“ sind nicht möglich. Die Transaktion B kann nur solche Daten lesen, die Transaktion A bereits per *commit()* dauerhaft an die Datenbank übergeben hat. Da während der Ausführung von B jedoch parallele Transaktionen die gleichen Daten mehrfach ändern können, erhält B beim wiederholten Lesen eines Tupels nicht unbedingt immer die gleichen Daten („Non-repeatable read“).
TRANSACTION_REPEATABLE_READ	„Dirty Reads“ und „Non-repeatable reads“ erfolgen nicht. Es kann aber passieren, daß Transaktion A ein neues Tupel in eine Tabelle einträgt, das Transaktion B beim wiederholten Lesen erhält („Phantom read“).
TRANSACTION_SERIALIZABLE	„Dirty reads“, „Non-repeatable reads“ und “Phantom reads“ erfolgen nicht. Transaktionen, die auf die gleichen Daten zugreifen, werden serialisiert, also nacheinander ausgeführt.

Die Datenbank arbeitet am schnellsten, wenn sich die Transaktionen nicht gegenseitig behindern, d.h. im Modus TRANSACTIONS_NONE. Maximale Datenintegrität wird nur durch den letzten Modus, TRANSACTION_SERIALIZABLE, gewährleistet. Der JDBC-Treiber initialisiert keinen dieser Modi, sondern verwendet die Voreinstellung der Datenbank.
Eine Änderung des Modus löst einen COMMIT auf der Datenbank aus. Während einer laufenden Transaktion darf der Modus demnach nicht geändert werden.
Informationen über die Art und Weise, wie die Datenbank Transaktionen unterstützt, können über die Methoden der Klasse DatabaseMetaData erhalten werden.

- supportsTransactions()
  Database Management Systems nach SQL2 Standard unterstützen auf jeden Fall Transaktionen.
- supportsMultipleTransactions()
  gibt an, ob ein DBMS auch über verschiedene Verbindungen mehrere Transaktionen gleichzeitig unterstützt.
- getDefaultTransactionIsolation()
  Welche Isolationsstufe ist standardmäßig für die Datenbank eingestellt?
- supportsTransactionIsolationLevel(n)
  liefert die Information, ob ein DBMS eine bestimmte Isolationsstufe n unterstützt.
- dataDefinitionIgnoredInTransactions(),
  dataDefinitionCausesTransactionCommit(),
  supportsDataDefinitionAndDataManipulationTransactions(),
  supportsDataManipulationTransactionsOnly()
  liefern Informationen darüber, welche Operationen bei Datendefinitionsanweisungen innerhalb einer Transaktion durchgeführt werden.

[181] Rainer Klute: JDBC in der Praxis, Addison-Wesley, S. 134

2.3.4.3.3 ORACLE Call Interface (OCI)

OCI

OCI ist die Programmierschnittstelle von Oracle für die Sprache C. Denkbar maschinennah, ist die OCI die von den Laufzeiten her schnellste Art, den Oracle Server anzusprechen. Hierüber werden direkt ORACLE-Unterprogramme aufgerufen, die in sprachspe-zifischen "run time"-Bibliotheken vorliegen. OCI ist ORACLE-Benutzern auch als High Level Interface (HLI) bekannt. Die grundlegende Programmstruktur eines OCI-Programms für eine SQL-Anfrage ist:

```
      OLON      "Einloggen" in das ORACLE-System unter Angabe
       |        von Passwort und Kennung
      OOPEN
       |        Öffnen des Cursors, der beim nächsten "OSQL3-
       |        Aufruf für die Aufnahme von Ergebnissen dient
      OSQL3
       |        Angabe des SQL-Kommandos
      ODEFIN
       |        Definition eines Puffers,in dem die C-Proramm-
       |        variablen angegeben werden, in dem die im
       |        SQL-Kommando abgefragten Werte gespeichert
       |        werden sollen. Für jedes Element des SELECT-
       |        Kommandos muß ein "ODEFIN"-Kommando angegeben
       |        werden.
      OBNDRV
      oder
      OBNDRN
       |        Definition aller C-Programmvariablen, die zur
       |        Übergabe von Daten aus dem C-Programm an die
       |        SQL-Abfrage verwendet werden sollen
      OEXE
       |        DurchfÜhrung einer SQL-Anfrage. Nach Durch-
       |        führung des "oexec"-Kommandos liegen die
       |        gefundenen Lösungen im angegebenen CURSOR-
       |        DATA-AREA bereit und können mit "OFETCH" ab-
       |        geholt werden
+--> OFETCH
|              Jeweils eine Zeile einer Tabelle wird gemäß
|nein          dem SQL-Ausfruf bereitgestellt
|
+--  EOF
       |
       |        Falls das Ende eines OFETCH noch nicht
                erreicht ist, wird der OFETCH-Aufruf zur
       ja       Bestimmung der nächsten Zeile ausgeführt
       |
       |        Falls alle Zeile bereitgestellt sind, folgt
       |        evtl. eine weitere SQL-Anweisung (Sprung nach
       |        OSQL3

       OCLOSE   Schließen des SQL-CURSOR
       |
       v
       OLOGOFF  Ausketten aus ORACLE
```

Bsp.: Ein OCI-Programm

```
/*=======================================================*/
/*                                                       */
/* Das Programm ermittelt Loesungen fuer die SQL-        */
/* Abfrage                                               */
/* Ermittle alle Angestellten, die in der Abteilung      */
/* 'Konstruktion' beschaeftigt sind.                     */
/*                                                       */
/* SELECT ANGESTELLTE.ID, ANGESTELLTE.NAME,              */
/*        ANGESTELLTE.GEBJAHR                            */
/*    FROM ANGESTELLTE, ABTEILUNG                        */
/*    WHERE ANGESTELLTE.ABT_ID=ABTEILUNG.ID AND          */
/*          ABTEILUNG.BEZ=:konst;                        */
/*                                                       */
/*=======================================================*/

#include <stdio.h>
```

```
/*=====================================================*/
/* Groesse der ORACLE-Kommunikations-Datenstrukturen    */
/*=====================================================*/

#define LOGON_DATA_SIZE 64
#define CURSOR_DATA_SIZE 64

/*=====================================================*/
/* Codes fuer ORACLE-Datentypen                         */
/*=====================================================*/

#define TYPE_INTEGER 3
#define TYPE_STRING  5


static int oci_error (erg, fct, cda, doterm)
int    erg;
char * fct;
char * cda;
int    doterm;
{
#ifdef VERBOSE
   printf ("fct \"%s\" Result %d %d\n", fct, erg, *(short *) &cda[12]);
#endif  /* VERBOSE */
   if ((erg != 0) && doterm)
      exit (1);
}

/*=====================================================*/
/* Beginn des Hauptprogrammes                           */
/*=====================================================*/

main ()
{

   /*=================================================*/
   /* Logon-Data-Area :                               */
   /* Ueber diesen Puffer findet die Kommunikation    */
   /* zwischen ORACLE und dem Anwenderprogramm statt  */
   /*=================================================*/

   char logon_data_area [LOGON_DATA_SIZE];

   /*=================================================*/
   /* Cursor-Data-Area:                               */
   /* Der Cursor dient als Aufnahmepuffer fuer Zeilen */
   /* der Loesungsmenge einer SQL-Abfrage             */
   /*=================================================*/

   char cursor_data_area [CURSOR_DATA_SIZE];

   /*=================================================*/
   /* Datenbereiche fuer ORACLE-Kennung und ORACLE-   */
   /* Passwort                                        */
   /*=================================================*/

   char uid [40];
   char psw [20];

   /*=================================================*/
   /* Hilfsgroessen                                   */
   /*=================================================*/

   int erg;
   short rlen;
   short rcod;
```

```
   /*=================================================*/
   /* Zielpuffer fuer die Ergebnisse einer einzelnen    */
   /* Zeile aus der Ergebnismenge einer SQL-Abfrage     */
   /*=================================================*/

   char angid  [4];
   char angnam [11];
   char anggeb [9];

/*=================================================*/
   /* Puffer fuer Uebergaben vom C-Programm aus an      */
   /* das ORACLE-System                               */
   /*=================================================*/

   char konst [255];

/*=================================================*/
   /* Einloggen in das ORACLE-System unter Angabe von   */
   /* Passwort und Kennung                            */
   /*=================================================*/

   strcpy (uid, "SAUER@T:rfhs1012:ora1");
   strcpy (psw, "SAUER");

   erg = olon (logon_data_area, uid, -1, psw, -1, 0);
   oci_error (erg, "olon", cursor_data_area, 1);

/*=================================================*/
   /* Oeffnen des Cursors, der beim naechsten osql3-    */
   /* Aufruf fuer die Aufnahme von Ergebnissen dienen   */
   /* soll                                            */
   /*=================================================*/

   erg = oopen (cursor_data_area,  logon_data_area, (char *) 0,
         -1, -1, (char *) 0, -1);
   oci_error (erg, "oopen", cursor_data_area, 1);

   /*=================================================*/
   /* Angabe des SQL-Kommandos, das beim nachsten oexec-*/
   /* Kommando auf die angegebene CURSOR-DATA-AREA      */
   /* durchgefuehrt werden soll                       */
   /*=================================================*/

   erg = osql3 (cursor_data_area,
               "SELECT ANGESTELLTE.ANG_ID, ANGESTELLTE.NAME, \
                       TO_CHAR( ANGESTELLTE.GEBJAHR ,'DD.MM.YY') \
                   FROM ANGESTELLTE, ABTEILUNG \
                 WHERE ANGESTELLTE.ABT_ID = ABTEILUNG.ABT_ID AND\
                       ABTEILUNG.BEZEICHNUNG = :konst", -1);
   oci_error (erg, "osql3", cursor_data_area, 1);


   /*=================================================*/
   /* Es muessen nun die C-Programmvariablen angegeben  */
   /* werden, in die die in dem SQL-SELECT-Kommando     */
   /* abgefragten Werte abgespeichert werden sollen.    */
   /* Fuer jedes Element des SELECT-Kommandos muss ein  */
   /* odefin-Kommando abgegeben werden                */
   /*=================================================*/

   erg = odefin (cursor_data_area, 1, angid,  sizeof (angid),
                 TYPE_STRING,
           -1, (short *) 0, (char *) 0, -1, -1, &rlen, &rcod);
   oci_error (erg, "odefin angid", cursor_data_area, 1);
   erg = odefin (cursor_data_area, 2, angnam, sizeof (angnam),
```

```
                TYPE_STRING,
           -1, (short *) 0, (char *) 0, -1, -1, &rlen, &rcod);
   oci_error (erg, "odefin angnam", cursor_data_area, 1);
   erg = odefin (cursor_data_area, 3, anggeb, sizeof (anggeb),
                TYPE_STRING,
           -1, (short *) 0, (char *) 0, -1, -1, &rlen, &rcod);
   oci_error (erg, "odefin anggeb", cursor_data_area, 1);

   /*=================================================*/
   /* Definition aller C-Programm-Variablen, die zur    */
   /* Uebergabe von Daten aus dem C-Programm an die     */
   /* SQL-Abfrage verwendet werden sollen               */
   /*=================================================*/

   erg = obndrv (cursor_data_area, ":konst", -1, konst, sizeof (konst),
         TYPE_STRING, -1, (short *) 0, (char *) 0, -1, -1);
   oci_error (erg, "obndrv konst", cursor_data_area, 0);


   /*=================================================*/
   /* Wertzuweisung an diese Uebergabevariablen         */
   /* (Hier z.B KONSTRUKTION, PERSONALABTEILUNG)        */
   /* (Gross- und Kleinschreibung muessen bei Strings   */
   /* beachtet werden)                                  */
   /*=================================================*/

   printf ("Bitte geben Sie die Abteilung an\n");
   gets (konst);
   printf ("konst : \"%s\"\n", konst);
      /* Muss mit Leerzeichen aufgefuellt sein */
/*   while (strlen (konst) < 40)
      strcat (konst, " ");
*/
   /*=================================================*/
   /* Durchfuehrung der SQL-Abfrage. Nach Durchfuehrung */
   /* des oexec-Kommandos liegen die gefundenen Loes-   */
   /* ungen im angegebenen CURSOR-DATA-AREA bereit und  */
   /* koennen mit ofetch abgeholt werden                */
   /*=================================================*/

   erg = oexec (cursor_data_area);
   oci_error (erg, "oexec", cursor_data_area, 1);

   do
   {

      /*=================================================*/
      /* Holen der naechsten Zeile aus der Loesungs-    */
      /* tabelle der mit dem angegebenen Cursor ver-    */
      /* bundenen SQL-Abfrage                           */
      /* Return-Code 4 : keine Loesung mehr vorhanden   */
      /*=================================================*/

      erg = ofetch (cursor_data_area);
      oci_error (erg, "ofetch", cursor_data_area, 0);


      /*=================================================*/
      /* Ggf. Ausgabe der gefundenen Loesung            */
      /*=================================================*/

      if ((erg >= 0) && (erg < 4))
        printf ("%s %s %s\n", angid, angnam, anggeb);

   } while ((erg >= 0) && (erg < 4));
```

```
   /*================================================*/
   /* Schliesen des SQL-Cursors                      */
   /*================================================*/

   erg = oclose (cursor_data_area);
   oci_error (erg, "oclose", cursor_data_area, 1);

   /*================================================*/
   /* Ausloggen aus dem ORACLE-System                */
   /*================================================*/

   erg = ologof (logon_data_area);
   oci_error (erg, "ologof", cursor_data_area, 1);
}
```

Oracle C++ Call Interface (OCCI)

OCCI ermöglicht den Zugriff auf den Oracle Server über eine in C++ programmierte Schnittstelle.

3. Objektrelationale Datenbanken

3.1 Objektrelationales SQL in SQL:99

3.1.1 Objektrelationales Typsystem

Das Typsystem von Standard-SQL bietet neben einer Reihe von Basisdatentypen auch eine Auswahl von Typkonstruktoren an:

- Basisdatentypen
- Tupeltypkonstruktor
- Arraytypkonstruktor
- Multimengentypkonstruktor
- Referenztypkonstruktor

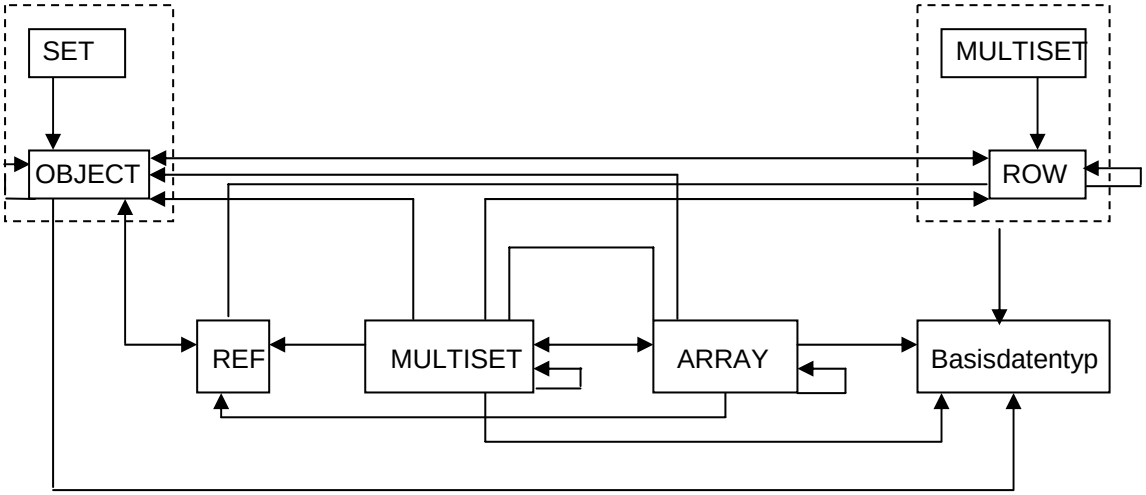


Abb.: Objektrelationale Datenbankmodell SQL:99

3.1.2 Datendefinition

Die Datendefinition in Standard-SQL umfaßt die Beschreibung folgender Schemaobjekte

- benutzerdefinierte Datentypen
- Tupeltabelle und Sichten
- benutzerdefinierte Routinen und Konstruktoren
- Trigger
- benutzerdefinierte Casts und Ordnungen

3.1.3 Anfragen

1. Anfragespezifikation

Die Spezifikation einer Anfrage in SQL besteht aus folgenden Klauseln:

SELECT *Projektionsliste*]
FROM *Tabellenreferenzliste*]
[**WHERE** *Selektionsbedingun*g]
[**GROUP BY** *Gruppierungsliste*]
[**HAVING** *Selektionsbedingung*]

2. Verbundoperationen

Folgende Verbundausdrücke werden unterstützt:

- das (explizite) Kreuzprodukt (CROSS JOIN)
- der innere Verbund, der passende Zeilen zweier Tabellen anhand einer Verbindungsbedingung verknüpft, die entweder mit der
  -- On-Klausel: ON Prädikat
  oder mit
  -- USING spaltennamensliste
  ausgedrückt wird.

3. Mengenoperationen

anfrageausdruck
{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT]
[CORRESPONDING [BY (*spaltennamenliste*)]]
anfrageausdruck

4. Sortierung

Die Order-by-Klausel

anfrageausdruck
ORDER BY sortierliste

ermöglicht die Sortierung in der Ergebnistabelle einer Abfrage

5. Navigierende Anfragen

Navigierende Ausdrücke werden mit Hilfe von Pfadausdrücken formuliert. Pfadausdrücke ergeben sich durch Schachtelung von Tupelfeldzugriffeb und Dereferenzierungen, z.B.:

```
select anr, vorgesetzter->anr
from angestellte
where anschrift.ort = vorgesetzter->anschrift.ort;
```

6. Anfrage mit Methodenaufrufen

Ein Methodenaufruf liefert einen Wert. Demnach kann eine Methode überall dort eingesetzt werden, wo ein Wert erwartet wird

7. Anfrage mit Kollektionen

Der UNNEST Operator „UNNEST (*kollektionswertausdruck*) [WITH ORDINALITY]“ wandelt eine Kollektion in eine Tabelle um

8. Anfrage auf flache Tabellenextensionen

Die flache Extension einer Tabelle enthält nur die Zeilen, die nicht einer ihrer Subtabellen angehören. Der Ausdruck `ONLY (tabellen-bzw-anfragenname)` liefert die flache Extension einer typisierten Tabelle.

9. Typspezifische Anfragen

Der Typ einer Instanz kann mit dem Typprädikat abgefragt werden:

`strukturierter-wertausdruck` IS [NOT] OF (*typnamensliste*)

Die Typnamenslsite besteht aus einem oder mehreren Typnamen.

10. Anfragen mit temporärer Typanpassung

Methoden werden immer mit dem speziellsten Typ aufgerufen. Der Standard ermöglicht innerhalb einer Anfrage bzw. Anweisung die Anpassung des speziellsten Typ einer Instanz eines strukturierten Typs entlang der Typhierarchie. Eine Anpassung an einen Supertyp geschieht mit dem folgenden Ausdruck:

strukturierter-wertausdruck as typname

Bsp.: Berechnung des Jahresgahalts der Manager mit der mitarbeiterspezifischen Jahresgehalsmethode

```
select (dref(oid) as angestellterTyp.jahresgehalt as angGehalt from
manager;
```

Eine Anpassung an einen Subtyp erfolgt mit dem TREAT-Ausdruck:

TREAT(*strukturierter-wertausdruck* as *typname*)

11. Anfrage auf Tabellenfunktionen

Mit dem Table-Konstrukt kann eine Tabellenfunktion als Tabellenreferenz in einer Anfrage verwendet werden, z.B.:

```
select name from table(besserverdiener(Euro(5000)));
```

12. Rekursive Anfragen

Eine rekursive Anfrage hat folgendes Grundgerüst:

WITH RECURSIVE *rekursionstabelle* (*spaltenliste*) AS
(
 -- Rekursionsinitialisierung
 SELECT
 FROM *tabelle*
 WHERE *initialisierungsprädikat*

```
 -- Rekursionsschritt
 UNION ALL
 SELECT
 FROM tabelle , rekursionstabelle
 WHERE rekursionsbedingung
)
[Traversierungsklausel][Zyklusklausel]
SELECT … FROM rekursionstabelle
```

3.1.4 Datenmanipulation

Das Einfügen, Ändern und Löschen von Tabellenzeilen erfolgt mit Hilfe der DML-Anweisungen: INSERT, UPDATE bzw. DELETE.

3.2 Objektrelationales SQL in Oracle

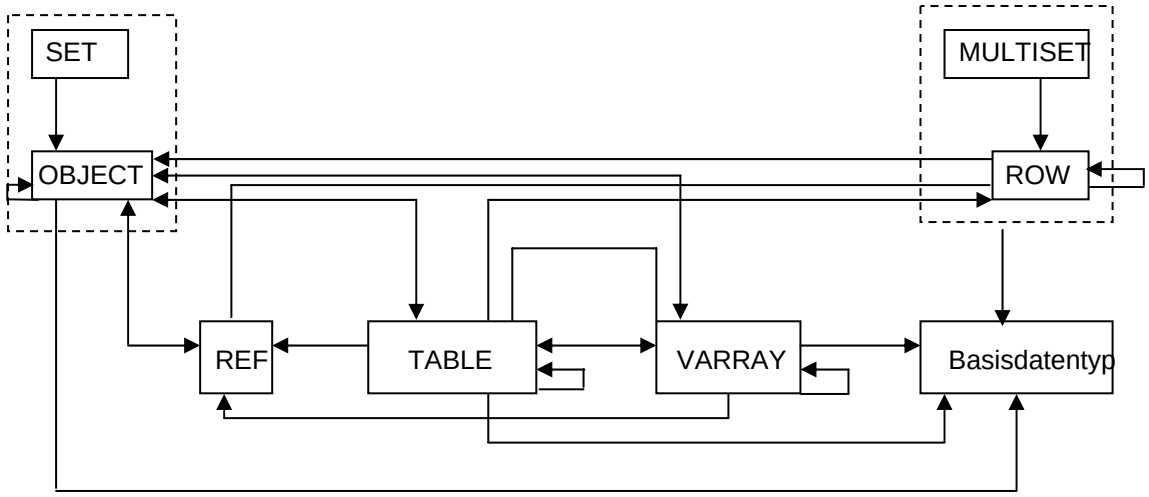


Abb.: Objektrelationales Datenbankmodell Oracle

3.2.1 Objektrelationales Typsystem

3.2.1.1 Neue Basisdatentypen

- Large Objects (LOBs)
  Hierbei handelt es sich um Datentypen, die es erlauben auch sehr große Attributwerte für z.B. Multimedia-Datenbanken zu speichern. Die Größe kann bis zu einigen Giga-Bytes betragen. Oracle unterscheidet zwei prinzipiell verschiedene Arten: internal LOBS und external LOBs (Binaray Files).

-- BLOB (LOB mit unstrukturierten Binärdaten)
-- CLOB (ein LOB, dessen Werte aus 1-Byte Character-Daten bestehen, entsprechend dem Character-Set, das für die Datenbank definiert wurde)
-- NCLOB (ein LOB, dessen Daten aus Character-Daten bestehen entsprechend einem nationalen Character-Set)
-- BFILE ("Externes" Large Object, Binary Files sind Verweise auf Betriebssystemdateien im Filesystem)

-- Operationen
--- EMPTY_BLOB, EMPTY_CLOB[182], BFILENAME
--- IS [NOT] NULL

-- Nicht erlaubt
--- BLOB/CLOB-Attribute
--- GROUP BY, ORDER BY

Basisdatentyp	Bedeutung
NUMBER(p,q)	Zahl mit p <= 38 und –84<=q<=127 [$1.0*10^{-130}$,9.9…$9*10^{125}$
SMALLINT	NUMBER(38)
INTEGER	NUMBER(38)
DECIMAL(p,q)	NUMBER(p,q)
NUMERIC(p,q)	NUMBER(p,q)

[182] EMPTY_BLOB() bzw. EMPTY_CLOB() initialisieren LOB-Lokatoren (notwendig für spätere Zuweisung)

FLOAT(p) REAL DOUBLE PRECISION	NUMBER mit Genauigkeit p (<= 126)
CHAR(q)	Alphanumerische Zeichenketten mit fester Länge (<= 2000)
VARCHAR(q)	Alphanumerische Zeichenketten mit variabler Länge (<= 4000)
CLOB	Alphanumerische Zeichenkette variabler Länge bis 4GB
BLOB	Binäre Zeichenkette mit variabler Länge bis 4 GB
BFILE	Lokator auf externe Datei der Grösse bis 4 GB
LONG RAW	Binäre Zeichenkette variabler Länge bis 2 GB
RAW	Binäre Zeichenkette mit variabler Länge bis 32 KB
DATE	Datum
TIMESTAMP	Zeitstempel
INTERVAL	Zeitintervall

Abb. Basisdatentypen

Internal LOBs
werden innerhalb des Datenbanksystems gespeichert und fallen somit unter das Transaktionskonzept. Damit ist also auch das recovery von LOB-Dateien im Fall von Systemfehlern möglich, Änderungen an LOB-Dateien können commited oder zurückgerollt werden.
Bei inserts und updates eines LOB-Attributes mit den Werten eines anderen LOB's werden komplette Werte kopiert, d.h. es wird nicht referenziert, die betreffenden Spalten enthalten jeweils die kompletten LOB-Daten.
CLOB- und NCLOB-Daten werden mit dem 2-Type Unicode für Character-Sets variabler Länge gespeichert. Beim Laden und Speichern aus und in CLOBs bzw. NCOLBs werden die Daten von speziellen Character-Set-Code in Unicode übersetzt und umgekehrt.
Eine Tabelle kann eine beliebige Anzahl von LOB-Attributen besitzen. LOB-Daten werden nicht im Inneren der Tabelle gespeichert, sondern außerhalb, an irgend einer anderen Stelle im Tablespace. Dies wird dadurch ermöglicht, dass in der entsprechenden Tabellenzeile ein sog. locator im LOB-Attribut gespeichert wird. Dieser locator ist ein Zeiger auf den aktuellen Speicherplatz der LOB-Daten.. Eine Besonderheit interner LOBs ist noch, LOB-Dateien, die weniger als 4 KB umfassen, können direkt in der zugehörigen Tabelle abgelegt sein.

External LOBs (BFILE)
sind große Binärdatenobjekte, die nicht im Datenbanksystem, sondern im Dateisystemsystem des Servers gespeichert sind. BFILEs unterliegen nicht dem Transaktionskonzept der Datenbank, insbesondere können also Änderungen der Daten nicht mit commit, rollback, recovery bearbeitet werden.
Bevor auf einen externen LOB (BFILE) zugegriffen werden kann, muß das BFILE-Attribut der Tabelle initialisiert werden (mit einem `locator` versehen werden). Dazu muß dem DBMS bekannt gemacht werden, wo im Dateisystem des Servers die Datei liegt. Dies geschieht durch:

```
create directory textaustausch as '/home3/bedienst/saj39122/oracle/text';
```

Directories können mit `drop directory` *`name`* wieder gelöscht werden.
Mit Hilfe eines existierenden directory kann ein BFILE-Attribut in einer Tabelle erzeugt werden unter Benutzung der Funktion BFILENAME():

```
BFILENAME(verzeichnisname IN varchar2, dateiname in varchar2)
```

Rückgabewert ist BFILE, also ein locator. BFILEs werden in Tabellenspalten über einen sog. locator referenziert.
BFILENAME kann in einer insert- oder update-Anweisung benutzt werden oder zur Initialisierung einer PL/SQL-Variablen. Wichtig ist, dass im Aufruf der Funktion BFILENAME der Name des `directory` in Großbuchstaben angegeben wird.
Bsp.:

```
create table progrText
(
 name varchar2(20),
 program clob,
 programmquelltext BFILE
);
insert into progrText values
(
 'ImageViewer.java',
 EMPTY_CLOB(),
 BFILENAME('TEXTAUSTAUSCH','ImageViewer.java')
);
```

Allgemein kann mit einer `delete`-Anweisung ein `locator` gelöscht werden, z.B.:

```
delete from progrtext where name = 'Imageviewer.java';
```

Die Verarbeitung von LOB-Daten kann mit verschiedenen Werkzeugen geschehen:

- Verarbeiten mittels PL/SQL
- Verarbeiten mit JAVA/JDBC

LOB-Verarbeitung mit PL/SQL

Oracle bietet zur Verarbeitung von LOB-Daten ein eigenes Package an: DBMS_LOB-Package. Das Package DBMS_LOB besteht aus einer Reihe von Funktionen zum Lesen und Modifizieren von internen bzw. externen LOBs.
Hat das BLOB- bzw. CLOB-Attribut einen `locator` (mit der ersten insert- oder update-Anweisung oder bereits mit der create-Anweisung über EMPTY_BLOB() bzw. EMPTY_CLOB() initialisiert), dann kann auf es zugegriffen werden, z.B. über PL/SQL

```
DECLARE
 textdaten CLOB;
 ...
 ...
begin
 select programm into textdaten
 from progrText
 where name = 'ImageViewer.java' for update;
-- jetzt kann mit dem locator-Wert von textdaten
-- manipuliert werden.
```

Mit dem DBMS_LOB-Package können LOB-Werte verändert bzw. ausgewählt werden.

Unterprogramme	Beschreibung
APPEND-Prozedur	hängt die Inhalte eines LOB-Werts an einen anderen LOB-Wert an.
CLOSE-Prozedur	schließt einen zuvor geöffneten internen oder externen LOB
COMPARE-Prozedur	vergleicht 2 LOB-Werte
COPY-Prozedur	kopiert den gesamten LOB oder bestimmte Teile in ein Ziel-LOB
ERASE-Prozedur	löscht den LOB ganz oder teilweise
FILECLOSE-Prozedur	schließt die Datei
FILEEXISTS-Funktion	Prüft, ob die Datei auf dem Server vorhanden ist

FILEGETNAME-Prozedur	Holt den Dateinamen und den Verzeichnis-Alias ab
FILEISOPEN-Prozedur	Prüft, ob eine Datei mit Hilfe des Eingabe-BFILE-Locator geöffnet w.
GETLENGTH-Funktion	holt die Länge des LOB-Werts ab
INSTR-Funktion	liefert die übereinstimmende POsition des n-ten Vorkommens des Musters im LOB
ISOPEN-Funktion	Prüft, ob das LOB bereits mit Hilfe des Eingabe-Locators geöffnet w.
LOADFROMFILE-Prozedur	Lädt BFILE-Daten in ein internes LOB
LOADBLOBFROMFILE-Proz.	Lädt BFILE-Daten in einen internen LOB
LOADCLOBFROMFILE-Proz.	Lädt CLOB-Daten in einen internen LOB
OPEN-Prozedur	Öffnet ein LOB im angegebenen Modus
READ-Prozedur	Liest die Daten des LOB ab dem vorgegebenen Offset
SUBSTR-Funktion	liefert Teile des LOB-Werts ab dem vorgegebenen Offset
TRIM-Funktion	kürzt den LOB-Wert auf die vorgegebene Länge
WRITE-Prozedur	schreibt die Daten ab dem vorgesehenen Offset an den LOB
WRITEAPPEND-Prozedur	schreibt einen Puffer an das Ende des LOB

Abb.: Prozeduren und Funktionen im DBMS_LOB-Package

Der folgende PL/SQL-Block lädt Textdokumente (Programmquelltext) aus der Tabelle `progrText`:

```
DECLARE
 textdaten CLOB;
 dok_datei BFILE:=BFILENAME('TEXTAUSTAUSCH','ImageViewer.java');
 loblaenge integer;
begin
 select programm into textdaten
 from progrText
 where name = 'ImageViewer.java' for update;
 DBMS_LOB.FILEOPEN(dok_datei,DBMS_LOB.LOB_READONLY);
 loblaenge := DBMS_LOB.GETLENGTH(dok_datei);
 DBMS_LOB.LOADFROMFILE(textdaten,dok_datei,loblaenge);
 DBMS_LOB.FILECLOSE(dok_datei);
 update progrText
 set programm = textdaten
 where name = 'ImageView.java';
 COMMIT;
end;
/
```

Über die Funktion READ() bietet das DBMS_LOB-Package die Möglichkeit an, LOB-Daten aus einem LOB-Attribut einer Tabelle auszulesen. DBMS_LOB.READ() hat 4 Parameter:

```
- Lob_loc IN                                    BLOB oder CLOB oder BFILE
- amount IN OUT                          BINARY INTEGER
-- die Anzahl der Bytes (BLOB) bzw. Character (CLOB), die gelesen werden sollen
- offset IN                                     INTEGER
-- das offset in BYTES (BLOB) bzw. Character (CLOB), von dem aus gelesen wird
- buffer OUT                                   RAW
-- Ausgabepuffer

set serveroutput on;
DECLARE
 textdaten CLOB;          -- locator-var
 puffer varchar2(500);    -- Variable fuer die Ausgabe
 textlaenge number;       -- Anzahl zu lesender Zeichen
 position number;
 read_counter number;
 amount number;           -- Variable fuer den Offset
begin
 position := 1;
```

```
 select programm into textdaten
 from progrText
 where name = 'ImageViewer.java';
 textlaenge := DBMS_LOB.GETLENGTH(textdaten);
 read_counter := 0;
 amount := 50;
 WHILE read_counter < textlaenge
 LOOP
    DBMS_LOB.READ(textdaten, amount, read_counter+1, puffer);
    DBMS_OUTPUT.PUT_LINE(puffer);
    read_counter := read_counter + amount;
 END LOOP;
end;
/
```

Innerhalb eines PL/SQL-Blocks kann das DBMS_OUTPUT-Package zum Anzeigen von Variablenwerten verwendet werden. PUT_LINE() zeigt die Ausgabe in einer Zeile. Vor dem Einsatz des DBMS_OUTPUT-Package ist der Befehl "`set serveroutput on`" auszuführen.

3.2.1.2 Aggregations-Typen

3.2.1.2.1 Variabler Arraytyp

- `create type` *`arraytypname`* `as varray(`*`arraygroesse`*`) of` *`elementtyp`*

-- Alle Elemente haben den gleichen Typ
-- Typ kann UDT (User Defined Type) sein
-- Maximallänge muß definiert werden. Mit der `modify limit` –Klausel des `alter type` – Befehls kann die maximale Anzahl der erlaubten Einträge in einem variablen Array verändert werden.
-- Zugriff über ganzzahligen Feldindex
-- Verwendung als Datentyp für Spalten
-- in Oracle: eigener Datentyp nötig[183]

Bsp.: `create type telefonArrayTyp as varray(30) of varchar(20);`
`/`
Nach Ausführung des Befehls gibt es einen Typ mit dem Namen "`telefonArrayTyp`". Die "`as varray(30)`"-Klausel teilt mit, dass ein variables Array angelegt wurde, und dieses Array pro Datensatz 30 Einträge enthalten kann.

3.2.1.2.2 Tabellentyp und verschachtelte Tabellen (nested tables)

Mit geschachtelten Tabellen lassen sich Zusammenhänge modellieren, die sonst in verschiedenen Tabellen abgelegt werden und in Anfragen über JOIN ausgewertet werden müssten. Eine solche Lösung bietet sich für mengenwertige Attribute oder 1:n-Beziehungen an, oft sogar für m:n-Beziehungen.
Geschachtelte Tabellen unterstützen keine (referentiellen) Integritätsbedingungen, die den Inhalt der inneren Tabelle mit dem umgebenden Tupel oder einer anderen Tabelle verknüpfen

- `create [or replace] type` *`name`* `as TABLE of` *`datatype`*
("Kollektion", Tabellen als Datentypen)

[183] SQL 99 erwähnt nur Collection-Typ ARRAY

TABLE: geordnete Multimenge mit indirektem Zugriff
- ohne Grössenbeschränkung
- alle Elemente besitzen denselben Typ
- Elementindex als Iterator
- Anfragen auf geschachtelte Tabellen anwendbar
- Daten der geschachtelten Tabelle sind auch indizierbar

Bsp.:
```
create type hobbiesTabellenTyp as table of varchar(20);
/
```

Verschachtelte Tabellen können dann so aufgebaut werden:

```
CREATE [OR REPLACE] TYPE inner_type
AS OBJECT ( ... );
/
CREATE [OR REPLACE] TYPE inner_table_type
AS TABLE OF inner_type;
/
CREATE TABLE table_name
( …..,
 table_attr inner_table_type,
 …. )
NESTED TABLE table_attr STORE AS name;
```

Beim Anlegen einer Tabelle, die eine verschachtelte Tabelle enthält, muß der Name der Tabelle angegeben werden, in der die Daten der verschachtelten Tabelle gespeichert werden. Daten der verschachtelten Tabelle werden nicht zusammen mit den Daten der Haupttabelle (also "inline") sondern separat gespeichert. Oracle unterhält zwischen den Tabellen so genammte Pointer oder Zeiger.
Bsp.:

```
create table mitarbeiterTupelTabelle (
name varchar(10) );
create type hobbiesTabellenTyp as table of varchar(20);
/
alter table mitarbeiterTupelTabelle
add hobbies  hobbiesTabellenTyp
nested table hobbies store as kundenhobbies;
```

In die Tabelle soll noch ein variabler Array aufgenommen werden

```
create type telefonArrayTyp as varray(30) of varchar(20);
/
alter table mitarbeiterTupelTabelle add telefone telefonArrayTyp;
desc mitarbeiterTupelTabelle
```

Mit Hilfe der ALTER-Anweisung kann (u.a.) das Datenbankschema verändert werden. Für alle Datenobjekte, die mit einem CREATE-Befehl erzeugt werden, gibt es den analogen DROP-Befehl, um sie zu löschen und den entsprechenden ALTER-Befehl zum Verändern des Objekts.
Durch Verwendung von Konstruktoren kann die Tabelle mit Daten ausgestattet werden

```
-- Erzeugen von Instanzen eines Arrays- bzw. Tabellentyps
insert into mitarbeiterTupelTabelle(name, hobbies, telefone)
values ('Johnny',
        hobbiesTabellenTyp('Reisen','Sport','Kino'),
        telefonArrayTyp('0041-1-6327248','0041-1-7337947'));
-- Zugriff auf Instanzen erfolgt analog zu anderen Attributen
select name, telefone, hobbies from mitarbeiterTupelTabelle;
select * from mitarbeiterTupelTabelle;
```

Bearbeitung innerer Tabellen

Tupel in innere Tabelle einfügen:
```
insert into TABLE
                        (<SFW_Query_liefert_innere_Tabelle>) values ( ...
                         )
```
Tupel einer inneren Tabelle ändern:
```
update
TABLE(<SFW_Query_liefert_innere_Tabelle>)
                           values ( ... )
                           set ...
                           where ... ;
```
Tupel einer inneren Tabelle löschen:
```
delete from
                           TABLE(<SFW_Query_liefert_innere_Tabelle>)
                           where ... ;
```

Einfügen, Aktualisieren und Löschen wird durch die TABLE-Funktion unterstützt. Die TABLE-Funktionn kann bei allen inserts und deletes eingesetzt werden, die direkt auf die verschachtelte Tabelle ausgeführt werden.

Bsp.:
```
insert into table (select hobbies
                   from mitarbeiterTupelTabelle
                   where name = 'Johnny') values('Film');
select * from mitarbeiterTupelTabelle;
update table (select hobbies
              from mitarbeiterTupelTabelle
              where name = 'Johnny') h
set value(h) = 'TV'      -- value ist hier noetig, um auf den Wert
where value(h) = 'Film'; -- der Tupel der inneren Tabelle
                         -- zuzugreifen
select * from mitarbeiterTupelTabelle;
delete table (select hobbies
              from mitarbeiterTupelTabelle
              where name = 'Johnny') h
where value(h) = 'Reisen';
select * from mitarbeiterTupelTabelle;
```

3.2.1.3 Abstrakte Datentypen

3.2.1.3.1 User Defined Types (Benutzerdefinierte Datentypen, Objekttypen)

Objektrelationale DBS unterstützen die Definition von anwendungsspezifischen Typen –oft **user defined types** (UDTs) genannt.
Mit CREATE TYPE kann eine neue Klasse von Schema-Objekten erzeugt werden:

```
- create [or replace] type name as OBJECT[184]
  ( attr datatype,
    attr datatype,
    ...
    attr datatype);
```

OBJECT: strukturierter (komplexer) Typ mit Methoden und Subtypbildung.

`CREATE TYPE typename AS OBJECT ( … )` definiert automatisch eine Konstruktormethode *typename*:

Bsp.:

[184] Bei echten Objekten kommt noch ein CREATE VIEW BODY ... dazu, in dem die Methoden in PL/SQL definiert werden. Ohne BODY bekommt man einfache komplexe Datentypen (ähnlich wie bei "records").

```
rem
rem User Defined Structured Types
rem
create or replace type adresse_t as object
(
 strasse varchar2(30),
 hausnr  number(3),      -- Basistyp
 plz     number(5),      -- Basistyp
 ort     varchar2(40)    -- Basistyp
);
/
desc adresse_t
create type personal_t as object
(
 nachname varchar2(20),
 vorname varchar(20),
 geburtsdatum date,
 gehalt number(7,2),
 kinder number(5),
 adresse adresse_t -- benutzerdefinierter Typ
);
/
select object_name from user_objects
where object_type = 'TYPE';
desc personal_t
-- Tabelle fungiert als Container fuer Objekte des angegebenen Typs
create table personal
(angestellter personal_t);
set describe depth 2
desc personal
-- Jeder benutzerdefinierte Datentyp besitzt einen Konstruktor
-- gleichen Namens,
-- mit dem Objekte bzw. "Instanzen" dieses Typs erzeugt werden
können,
-- z.B.
insert into personal values
(
 personal_t('Mustermann',
            'Gabi',
            '07.08.1971',
            2500.00,
            2,
            adresse_t('Musterallee',1,12345,'Musterstadt')
           )
);
-- Selektion von Attributen benutzerdefinierter Typen ("Kaskadierte
-- Punktnotation"):
select p.angestellter.nachname, p.angestellter.adresse.ort
from personal p
where p.angestellter.gehalt > 2000.00;
drop table personal;
drop type personal_t;
drop type adresse_t;
desc user_objects
```

3.2.1.3.2 User Defined Methods

Eine Typ-Deklaration kann Methoden enthalten. Die Deklaration erfolgt ueber MEMBER FUNCTION bzw. MEMBER PROCEDURE in der CREATE TYPE - Anweisung.

```
CREATE TYPE typname AS OBJECT
(
```

```
 -- Atributdefinitions_und_Methodendeklarationsliste

 attribut datentyp,
 .
 .
 attribut REF datentyp,
 .
 .
 MEMBER FUNCTION funktionsname [(parameterlieste)] RETURN datentyp,
 .
 .
 MEMBER PROCEDURE prozedur-name [(parameterliste)],
 .
 .

 -- Abbildungs_und_Ordnungsmethode

 [ MAP MEMBER FUNCTION funktionsname RETURN datentype, |
   ORDER MEMBER FUNCTION funktionsname (variable typ) RETURN datentyp,]
 [ pragma-deklarations-liste ]
)
 [[NOT] FINAL]
[[NOT] INSTANTIABLE]
```

Default: FINAL und Integritätsbedingungen werden nicht unterstützt
OBJECT: Strukturierter Typ mit Methoden und Subtypbildung
(Defaultwerte und Integritätsbedingungen werden nicht unterstützt)
Methodendeklaration
[[NOT] OVERRIDING
[[NOT] FINAL
[[NOT] INSTANTIABLE
DEFAULT: NOT OVERRIDING, NOT FINAL, INSTANTIABLE
{MEMBER | STATIC}
{FUNCTION | PROCEDURE}
methodenName [(<*parameterliste*>)]
OVERRIDING: überschreibende Methode
FINAL
INSTANTIABLE
MEMBER: Instanzmethode wird auf einem Objekt aufgerufen (besitzt impliziten SELF-Parameter)
STATIC: statische Methode wird auf dem Objekttyp aufgerufen
PROCEDURE: hat keinen Rückgabewert
FUNCTION: hat einen Rückgabewert, darf aber keine Objektattributwerte ändern
PRAGMA-Klauseln. In pragma-deklarationsliste kann für jede Methode eine PRAGMA-Klausel der Form PRAGMA RESTRICT_REFERENCES (methoden-name, feature-liste) angegeben werden, die spezifiziert, ob eine Prozedur / Funktion lesend / schreibend auf die Datenbank zugreifen darf. feature-liste ist ein Komma-Liste mit den möglichen Einträgen:

WNDS	Writes no database state
WNPS	Writes no package state
RNDS	Reads no database state
RNPS	Reads no package state

Diese Angabe ist insbesondere für Funktionen wichtig, da Oracle diese nur ausführt, wenn zugesichert ist, dass sie den Datenbankzustand nicht verändern. Daher muß bei Funktionen zumindest PRAGMA RESTRICT_REFERENCES (funktionsname,WNPS,WNDS) gesetzt werden. MAP- und ORDER-Funktionen PRAGMA RESTRICT_REFERENCES (funktionsname,WNDS,WNPS,RNPS,RNDS), d.h. sie dürfen keinen Datenbankzugriff enthalten.

Bsp.: Gegeben ist

```
create type punktType as object
( x number,
  y number
```

```
);
/
desc punktType
-- Ein Objekttyp kann in einem anderen Objekt-Typ
-- verwendet werden
create type linieType as object
(
 end1 punktType,
 end2 punktType
);
/
desc linieType
-- Dieser Typ kann wiederum in einer Tabelle angesprochen werden
create table linien
(
 linienID int,
 linie linieType
);
desc linien
insert into linien
values (27, linieType(
             punktType(0.0,0.0),
             punktType(3.0,4.0)
                       )
        );
```

Der vorliegende Typ soll ergänzt werden durch

```
-- Hinzufügen einer Funktion laenge zum linienType.
-- Beim Erzeugen der Laenge wird mit einem Skalenfaktor multipliziert

create or replace type linieType as object
(
 end1 punktType,
 end2 punktType,
 member function laenge(skala in number) return number,
 pragma restrict_references (laenge, wnds)
);
/
desc linieType
```

Methoden werden anschliessend in einer **create type body** ... -Anweisung definiert. Der TYPE BODY enthält die Implementierung in PL/SQL. Für jedes Objekt ist automatisch die Methode SELF definiert, mit der das Objekt als Host-Objekt einer Methode angesprochen werden kann. SELF wird in Methodendefinitionen verwendet, um auf die Attribute des Host-Objekts zuzugreifen.. Die Definition des TYPE BODY muß der bei CREATE TYPE vorgegebenen Signatur desselben Types entsprechen. Insbesondere muß für alle deklarierten Methoden eine Implementierung angegeben werden.

```
CREATE [OR REPLACE] TYPE BODY type AS
MEMBER FUNCTION funktionsname [(parameterliste)] RETURN datentyp IS
 [variablen-deklarationsliste]
 BEGIN pl/sql-code END;
 .
 .
MEMBER PROCEDURE prozedur-name [(parameter-liste)] IS
 [variablen-deklarationsliste]
 BEGIN pl/sql-code END;
 .
 .
[MAP MEMBER FUNCTION funktionsname RETURN datentyp |
 ORDER MEMBER FUNCTION funktionsname (variable typ) RETURN datentyp
```

```
 IS
 [variablen-deklarationsliste]
 BEGIN pl/sql-code END; ]
END;
/
```

Im vorliegenden Bsp. ist der TYPE BODY für die Objektmethode `laenge`:

```
create type body linieType as
member function laenge(skala number) return number is
begin
 return skala *
         SQRT((SELF.end1.x - SELF.end2.x) *
              (SELF.end1.x - SELF.end2.x) +
              (SELF.end1.y - SELF.end2.y) *
              (SELF.end1.y - SELF.end2.y)
             );
end;
end;
/
```

Auf Werte der Komponenten von einem Objekt wird mit der Punktnotation zugegriffen
Methoden eines Objekts werden mit `objekt.methoden-name(argumentenliste)` aufgerufen.

```
select linienID, l1.linie.laenge(2.0)
from linien l1;
```

Wichtig ist hier der Alias-Name.

```
select l1.linie.end1.x, l1.linie.end1.y
from linien l1;
select ll.linie.end2
from linien ll;
```

ORDER- und MAP-Methoden. Objekttypen besitzen im Gegensatz zu den Datentypen NUMBER und VARCHAR keine inhärente Ordnung. Man unterscheidet:

- Systemdefinierte Gleichheit von Instanzen der Objekttypen. Sie basiert auf paarweiser Gleichheit der Attributwerte (Flache Gleichheit).
- Benurtzerdefinierte Gleichheit bzw. Ordnung über
  - MAP: Vergleich bzw. Ordnung basiert auf dem Ergenis einer Abbildungsfunktion, die Objektwerte auf die Werte von Basisdatentypen (Zahlen, Zeichenketten, Datum) abbildet.

    MAP MEMBER FUNCTION funktionsname RETURN typ
  - ORDER: Ordungsfunktion ordnet jeweils 2 Instanzen

    ORDER MEMBER FUNCTION funktionsname(o objekttyp) RETURN INTEGER

Um eine Ordnung auf Objekten eines Typs zu definieren, können dessen funktionale Methoden verwendet werden. Für jeden Objekttyp kann eine Funktion als MAP- oder ORDER-Funktion ausgezeichnet werden:

- Eine MAP-Funktion besitzt keine Parameter und bildet jedes Objekt auf eine Zahl ab. Damit wird auch eine lineare Funktion auf dem Objekttyp definiert, die sowohl für Vergleiche <, > und BETWEEN als auch für ORDER BY verwendet werden kann.
- Eine ORDER-Funktion besitzt ein Argument desselben Datentyps das ein reiner IN-Parameter ist und mit dem Hostobjekt verglichen wird. Eine ORDER-Funktion entspricht

einem direkten Vergleich zweier Werte / Objekte. Damit sind ORDER-Funktionen für Vergleiche <, > geeignet, im allgemeinen aber nicht unbedingt für Sortierung.
- MAP- und ORDER-Funktionen erfordern PRAGMA RESTRICT_REFERENCES(name,WNDS,WNPS,RNPS,RNDS), d.h. sie dürfen keinen Datenbankzugriff enthalten.

ORDER/MAP-Methoden definieren eine Ordnung auf Objekten eines Typs, und können damit auf zwei Arten verwendet werden:

- Ordnen nach dem Attributwert einer objektwertigen Spalte
- die Elemente einer Objekttabelle sollen nach ihrem eigenen Objektwert geordnet werden

Bsp.:

```
create or replace type adressTyp as object
(
 strasse varchar2(30),
 nr      decimal(4),
 plz     decimal(5),
 ort     varchar2(40),
 land    varchar2(25),
 MAP MEMBER FUNCTION adressMap RETURN varchar2
 -- alternativ koennte folgende ORDER-Funktion definiert werden
 -- ORDER MEMBER FUNCTION adressOrder(a adressTyp) RETURN INTEGER
) NOT FINAL;
/
-- Implementierung MAP-Funktion
create type body adressTyp as
MAP MEMBER FUNCTION adressMap return VARCHAR2
as
begin
 return land || ort || plz || strasse || nr;
end;
end;
/
-- ein Teil referenziert ein anderes Teil
create type teiltyp as object (
 nr number(10),
 bezeichnung varchar(25),
 farbe varchar(15),
 istTeilvon REF teiltyp
);
/
create type positionsTyp as object
(
 posnr integer,
 teil REF teiltyp,
 menge integer,
 preis integer
);
/
create type positionsTabellentyp
as table of positionsTyp;
/
create or replace type auftragTyp as object
(
 anr integer,
 lieferant REF lieferantTyp,
 eingang date,
 bearbeitet date,
 position positionsTabellenTyp,
 member function anzahl return integer
);
/
-- Methodenimplementierung
```

```
create type body auftragTyp as
 member function anzahl return integer
as
 a integer;
begin
 for i in 1..SELF.position.COUNT LOOP
  a := a + SELF.position(i).menge;
 END LOOP;
 return a;
end;
end;
/
create type telefonArrayTyp as varray(30) of varchar(20);
/
create type auftragTabellenTyp as table of REF auftragTyp;
/
create table kundeTupelTabelle
(
knr integer,
name varchar(30),
anschrift adressTyp,         -- objektwertiges Attribut
telefone telefonArrayTyp,    -- arraywertiges Attribut
auftraege auftragTabellenTyp –- kollektionswertiges Attribut
)
nested table auftraege store as kundenauftraege;
insert into kundeTupelTabelle
values
(15,
 'Wilhelm',
 adressTyp('Seefeldstrasse',31,8008,'Zuerich','CH'),
 telefonArrayTyp('0041-1-6789322','0049-6194-91267'),
 auftragTabellenTyp(NULL));
-- Verwendung von Map- bzw. Order-Funktion
select name, anschrift
from kundeTupelTabelle
order by anschrift;
select name, anschrift
from kundeTupelTabelle
where Anschrift < adressTyp('Seefeldstrasse',31,8008,'Zuerich','CH');
```

Operationen auf Objekttypen

Defaultkonstruktor heißt wie der Objekttyp und hat für jedes Attribut einen Parameter	objekttypname(p1, p2, ... , pn)
Attributzugriff mittels Punktnotation	o.Attributname)
Objektvergleich basiert auf flacher Gleichheit	o1 = o2
Referenzvergleich ist auch möglich	REF(o1) = REF(o2)
Typtest mittels IS OF	IS OF (objekttypname)

Abb.: Operationen auf Objekttypen

Subtypbildung – Aufbau von Typhierarchien.

Syntax: CREATE TYPE *subtypname* UNDER *supertypname*
(Attributdefnitions- und Methodendeklarationsliste
  [Überschreibende-Abbildungsmethode]
)
[[NOT] FINAL]
[[NOT] INSTANTIABLE]

- Subtyp erbt alle Attribute und Methoden des Supertyps
- Subtyp hat nur einen Supertyp (keine Mehrfachvererbung)
- Geerbete Methoden sind überschreibbar

Bsp.:

```
-- Supertyp
create or replace type kundeTyp as object
(
 knr integer,
 name varchar(30),
 anschrift adressTyp,
 telefone telefonArrayTyp,
 auftraege auftragTabellenTyp
) NOT FINAL;
/
create type hobbiesTabellenTyp as table of varchar(20);
/
-- Subtypbildung - Aufbau von Typhierachien
create type bwKundeTyp UNDER kundeTyp
(
 hobbies hobbiesTabellenTyp,
 kredit decimal(9,2)
);
/
```

3.2.1.4 Referenz-Typ

Für jeden Typ t ist REF t der Typ von Referenzen (Object IDs) auf Werte vom Typ t. Dieser Typ kann anstelle eines benutzerdefinierten Typs verwendet werden.

- Attribute können direkte Referenzen auf Tupel / Objekte (derselben oder anderer Relationen) als Wert haben
- Dadurch ist man nicht mehr auf die Nutzung von Fremdschlüsseln zur Realisierung von Beziehungen beschränkt.
- Insbesondere kann ein Attribut auch eine Menge von Referenzen als Wert haben, so dass man auch n:m Beziehungen ohne separate Beziehungsrelation repräsentieren kann.
- Referenzen setzen voraus, dass man Objekte (Tupel) anhand einer unveränderbaren Objektidentität (OID) eindeutig identifizieren kann
- Referenzen führen unvermeidbar zur Notwendigkeit, Pfadausdrücke in der Anfragesprache zu unterstützen.
- REF: Referenz auf eine Instanz eines Objekttyps
- Wert eines Referenzattributs ist eine OID

Bsp.:

```
create type punktType as object
( x number,
  y number
);
/
-- Typreferenzen
create table linien2
(
 end1 REF punktType,
 end2 REF punktType
);
```

3.2.1.5 Objekttabellen und OIDs

Objektorientierung in Oracle bedeutet: Es gibt neben Tabellen, deren Inhalt aus Tupeln besteht, auch Object Tables (Objekttabellen), deren Inhalt aus Objekten besteht. Ebenso können einzelne Spalten einer Tabelle objektwertig sein. Im Gegensatz zu einem Tupel besitzt ein Objekt Attribute, die den inneren Zustand

eines Objekts beschreiben, sowie Methoden, mit denen der Zustand abgefragt und manipuliert werden kann. Komplexe Attribute sind Objekttypen, die nur Wertattribute und keine Methoden besitzen. Objekte können neben Wertattributen auch Referenzattribute besitzen. Referenzattribute werden durch `REF objekt-datentyp` gekennzeichnet:

```
referenz-attribut REF objekt-datentype
```

Falls ein referenzierte Objekttyp in verschiedenen Tabellen vorkommt, wird damit das Zeil der Referenz nicht auf eine bestimmte Referenz beschränkt.
Nur Objekte, die ein OID besitzen können referenziert werden. Die OID eines zu referenzierenden Objekts wird folgendermaßen selektiert:

```
SELECT .... , REF (variable)
FROM tabelle variable
WHERE … ;
```

Zeilen- und Spaltenobjekte

Zeilenobjekte sind Elemente von Objekttabellen. Spaltenobjekte erhält man, wenn ein Attribut eines Tupels (oder eines Objekts) objektwertig ist. Zeilenobjekte erhalten eine eindeutige OID, über die sie referenzierbar sind. Spaltenobjekte haben keine OID und sind nicht referenzierbar.
Zeilen- und Spaltenobjekte werden mit Hilfe entsprechender Konstruktormethoden erzeugt. Für einen n-stelligen Konstruktor müssen n Argumente angegeben werden. Wenn die Werte zum Zeitpunkt der Objekterzeugung noch nicht bekannt sind, müssen NULL-Werte angegeben werden.
Bei SELECT-Anweisungen müssen immer Tupel- bzw. Objektvariable durch Aliasing verwendet werden, wenn ein Zugriffspfad `variable.methode[.attribut]` angegeben wird, d.h. wenn Attribute oder Methoden von objektwertigen Attributen ausgewählt werden.
Spaltenobjekte erhält man, indem man ein Attribut einer Tabelle (Tupel- oder Objekttabelle) objektwertig definiert. Auch komplexe Attribute definieren Spaltenobjekte.
Zeilenobjekte. Objekttabellen enthalten im Gegensatz zu relationalen Tupeltabellen keine Tupel sondern Objekte. Innerhalb jeder Objekttabelle besitzt jedes Objekt eine eindeutige OID (Object-ID), die dem Primärschlüssel im relationalen Modell entspricht. Dieser wird mit den weiteren Integritätsbedingungen bei der Tabellendefinition angegeben. Fremdschlüsselbedingunen können in der gewohnten Syntax angegeben werden:

Eine Objekttabelle

- basiert auf einem Objekttyp und speichert je Zeile eine Instanz des Objekttyps
- kann Integritätsbedingungen enthalten
- muß für jedes tabellwertiges Attribut je eine STORE-Klausel enthalten
- Subtabellenbildung ist nicht möglich
- Syntax

  CREATE tabellenname OF *objekttypname*
  [Substitutionsklausel]
  [(Integritätsbedingungsdefinitionsliste)]
  [OID-Klausel]
  [STORE-Klausel]
  [Attributsubstitutionsklausel]
- Die OID-Klausel legt die Art der Referenzgenerierung fest

- Systemgeneriert oder vom Primärschlüssel abgeleitet

OBJECT IDENTIFIER is {SYSTEM GENERATED | PRIMARY KEY}

- Eine Objekttabelle darf auch Instanzen eines Subtyps aufnehmen (Prinzip der Substituierbarket)

NOT SUBSTITUTABLE AT ALL LEVELS

- Attributsubstitutionsklausel schließt Substitution eines objektwertogen Attributs aus

COLUMN attributname
{NOT SUBSTITUTABLE AT ALL LEVELS | IS OF (ONLY attributname)}

Bsp.:

```
create table kunde of kundeTyp
(
 knr primary key,
 name not null,
 anschrift not null
 -- CHECK(anschrift.plz is not null)
)
nested table auftraege store as kundenAuftrag
column anschrift is of (only adressTyp);
-- Beispiel fuer eine Objekttabelle, die nur Instanzen eines Subtyps
-- aufnimmt
create table bwKunde of bwKundeTyp
NOT SUBSTITUTABLE AT ALL LEVELS
(
 knr primary key,
 name not null,
 anschrift not null,
 CHECK(anschrift.plz is not null),
 CHECK(kredit > 0)
)
nested table auftraege store as bwKundenauftraege
nested table hobbies store as bwKundenhobbies;
```

Operationen auf Objekttabellen

```
insert into kunde
values insert into kunde
values
( kundeTyp ( 34, 'Johnny',
             adressTyp('Fifth Avenue',45,45666,'BigTown','USA'),
             telefonArrayTyp('0041-1-6725655','0049-454-364666'),
             auftragTabellenTyp(NULL)
           )
)
```

"kunde" ist eine Objekttabelle. Beim Einfügen von Daten können die Konstruktormethoden des Datentyps eingesetzt werden.
Beim Einfügen einer Zeile in eine Objekttabelle weist Oracle jeder Zeile eine Object-ID (OID) zu. Mit dieser OID kann die Zeile als referenzierbares Objekt eingesetzt werden.

```
update kunde
set anschrift = adressTyp('Kreuzstrasse',21,8008,'Zuerich','CH')
where name = 'Johnny';
select * from kunde;
delete from kunde k
-- Korrelationsvariable k ist notwendig, um auf Objektattribute
-- zugreifen zu koennen
where k.anschrift.ort = 'Zuerich';
```

Basiert die Objekttabelle auf einem abstrakten Datentyp, der keine anderen Datentypen verwendet, hat ein update- oder delete-Befehl auf eine Objekttabelle das gleiche Format wie bei einer relationalen Tabelle.

Substituierbarkeit (Erzeugen eines Kunden mittels explizitem Konstruktoraufruf) Objekte werden unter Verwendung des Objekt-Konstruktors *objekttypname* in Objekttabellen eingefügt.
Bsp.:

```
insert into kunde
values insert into kunde
values
( kundeTyp ( 34, 'Johnny',
             adressTyp('Fifth Avenue',45,45666,'BigTown','USA'),
             telefonArrayTyp('0041-1-6725655','0049-454-364666'),
             auftragTabellenTyp(NULL)
           )
)
```

Substituierbarkeit (Erzeugen eines bwKunden mittels explizitem Konstruktoraufruf)
Bsp.:

```
insert into kunde
values (bwKundeTyp(23,
                   'Jim',
                   adressTyp('Fifth
Avwenue',45,45666,'BigTown','USA'),
                   telefonArrayTyp('0041-1-6761256'),
                   auftragTabellenTyp(NULL),
                   hobbiesTabellenTyp(NULL),
                   10000));
select * from kunde;
```

Die Funktionen REF, DEREF und VALUE in Objekttabellen. In einer Objekttabelle ist jede Zeile ein Zeilenobjekt. Eine Objekttabelle unterscheidet sich von einer relationalen Tabelle durch

- Jede Zeile besitzt innerhalb einer Objekttabelle eine OID, die beim Anlegen der Tabelle von Oracle zugewieden wird.
- Zeilen einer Objekttabelle können von anderen Objekten innerhalb der Datenbank referenziert werden

Bsp.:

```
-- Objekttyp mitarbeiter_t
create or replace type mitarbeiter_t as object
(
 mitarb_id varchar2(3),
 name      varchar2(15),
 gebdatum  date
);
/
-- Objekttabelle mitarbeiter
create table mitarbeiter of mitarbeiter_t;
-- Aufnehmen von Daten
insert into mitarbeiter values
( mitarbeiter_t('A1','Fritz','13.11.1956'));
```

Die Selektion von Daten aus der Objekttabelle `mitarbeiter` unterscheidet sich nicht von der Selektion von Daten aus Tupeltabellen.
Die REF-Funktion ermöglicht das Referenzieren von vorhandenen Zeilenobjekten:

```
select REF(m)
```

```
from mitarbeiter
where name = 'Fritz';
```

Die Ausgabe zeigt den OID für das Zeilenobjekt.
Die DEREF-Funktion nimmt einen Referenzwert (den generierten OID-Wert für eine Referenz) und liefert den Wert des Zeilenobjekts zurück.
Bsp.: Zur Demonstration der Zusammenarbeit von REF und DEREF soll folgende Tabelle betrachtet werden

```
create table inhaber
(
 inhaber_name varchar2(20),
 mitarbeiter_beschaeftigung REF mitarbeiter_t
 -- referenziert Spalten, die irgendwo anders gespeichert sind
);
```

Die Spalte `mitarbeiter_beschaeftigung` kann auf Zeilenobjekte in der Objekttabelle `mitarbeiter` zeigen. In die Tabelle inhaber können Datensätze eingefügt werden, z.B.:

```
insert into inhaber
select 'Katherina', REF(m)
from mitarbeiter m
where name = 'Fritz';
```

Mit

```
select * from inhaber;
```

kann die Referenz sichtbar gemacht werden. Die Spalte mitarbeiter_beschaeftigung enthält die Referenz auf das Zeilenobjekt (und nicht den Wert der Daten, die in den Zeilen gespeichert sind). Den referenzierten Wert kann man erst nach Einsatz der DEREF-Funktion sehen

```
select DEREF(i.mitarbeiter_beschaeftigung)
from inhaber i
where inhaber_name = 'Katherina';
```

Bei dieser Abfrage gibt es Hinweise, die Unterschiede zwischen Abfragen in relationalen Tabellen und Objekttabellen:

- Um von einer Tabelle (`inhaber`) zu einer zweiten Tabelle (Objekttabelle `mitarbeiter`) zu gelangen, benutzt die Abfrage eine Referenz auf ein Zeilenobjekt. Dabei wird im Hintergrund ein Join ausgeführt, ohne dass irgendwelche Join-Kriterien angegeben wurden.
- Die Objekttabelle wird in der Abfrage nicht erwähnt. Die einzige Tabelle, die in der Abfrage angeführt wird, ist inhaber. Um Werte über DREF zu erhalten, braucht man den Namen der Objekttabelle nicht zu kennen.
- Das vollständige referenzierte Zeilenobjekt wird zurückgeliefert, nicht nur Teile der Zeile.

Mit

```
select * from mitarbeiter;
```

kann bspw. die Objekttabelle abgefragt werden. Obwohl `mitarbeiter` eine Objekttabelle ist, kann auf die Tabelle, genau wie bei einer relationalen Tabelle, ein select ausgeführt werden (konsistent mit den vorstehenden selects und

inserts). Um bei der `mitarbeiter`-Tabelle die gleiche Struktur wie bei DEREF zu sehen, ist die VALUE-Funktion anzuwenden:

```
select value(m)
from mitarbeiter m
where name = 'Fritz';
```

Ungültige Referenzen. Man kann das Objekt löschen, auf das eine Referenz zeigt, z.B.:

```
delete from mitarbeiter
where name = 'Fritz';
```

Der Datensatz in `inhaber`, der auf den Datensatz in `mitarbeiter` verweist, besitzt einen sog. hängenden REF (dangling references). Falls für "Fritz" ein neuer Datensatz eingefügt wird, wird dieser Datensatz nicht als Bestandteil der Referenz erkannt, die zu einem früheren Zeitpunkt eingerichtet wurde. Entstandene "dangling references" lassen sich durch `WHERE ref-attribut IS DANGLING` überprüfen. Die kann bspw. in einem AFTER-Trigger der Form

```
update tabelle
 set attribut = NULL
 where attribut is dangling
```

überprüft werden.

3.2.1.6 Objekt-Views

Eine Objektsicht ist eine abgeleitete, virtuelle Tabelle:

- kann Daten aus mehreren Tabellen erhalten
- objekterhaltende Sicht ist nicht auf die flache Extension einer Objekttabelle beschränkt

```
CREATE VIEW sichtenname OF objekttyp
[WITH OBJECT IDENTIFIER (attributliste)]
AS select-anweisung
[WITH { CHECK OPTION | READ ONLY }
```

- objektgenerierende Sicht muß WITH OBJECT IDENTIFIER Klausel enthalten. Bei der Definition von Objektviews wird durch `WITH OBJECT OID` *`attr-liste`* angegeben, aus welchen Attributen die Objekt-ID berechnet wird.
- CHECK OPTION: geänderte Daten dürfen nicht aus der Objektsicht verschwinden
- READ ONLY: nicht änderbare Sicht
- *select-anweisung* darf kein Objektkonstruktor enthalten

Objekterhaltung (Objektsicht basierend auf Objekttabelle)

```
create view zkKunde of kundeTyp
as
(select * from kunde k
 where value(k) is of (only kundeTyp) and k.anschrift.ort = 'BigTown');
show errors;
select * from zkKunde;
```

Objektgenerierung (Objektsicht basierend auf Tupeltabelle)

```
create view zkKundeTupel of kundeTyp
WITH OBJECT IDENTIFIER (knr) as
```

```
(select knr, name, anschrift, telefone, auftraege from kundeTupelTabelle
k
    where k.anschrift.ort = 'BigTown');
select * from zkKundeTupel;
```

Eine objektgenerierende Sicht kann auch auf einer Objekttabelle generiert sein.

Subsichten

```
CREATE VIEW subsichtenname OF subtyp
UNDER supersichtenname
AS anfrageausdruck
[WITH {CHECK OPTION | READ ONLY }]
```

- Typ der Subsicht muß ein direkter Subtyp des Typs der Supersicht sein
- Extension der Subsicht ist immer eine Untermenge der Extension der Supersicht
  - Subsicht erweitert die Extension der Supersicht
- Subsicht hat genau eine direkte Supersicht
  - keine direkte Mehrfachspezialisierung möglich
- Jede Objektsicht darf pro Subtyp maximal eine Subsicht haben

Implementierung von Objekt-Views (Überlagern bereits vorhandener Tabellen mit objektorientierten Strukturen). Eine relationale Tabelle wurde mit folgenden Anweisungen erzeugt bzw. mit Daten gefüllt.

```
create table angestellte_basis
(
 ang_id varchar2(3),
 name varchar2(10),
 gebdatum date,
 beruf varchar2(30),
 gehalt number(8,2),
 einstelldatum date
);
-- Einfügen von Daten
insert into angestellte_basis
values
('A20','Horst','13.04.1976','Systemplaner',5000.00,'15.03.2002');
```

Die Datentypen `person_t` bzw. `job_t` werden angelegt

```
create or replace type person_t as object
(
 name varchar2(10),
 gebdatum date
);
/
-- Anlegen des Datentyps job_t
create or replace type job_t as object
(
 beruf varchar2(30),
 gehalt number(8,2),
 einstelldatum date
);
/
```

und in der Objekt-View `angestellte_ov` eingesetzt

```
create view angestellte_ov(ang_id, person, job) as
select ang_id,
       person_t(name,gebdatum),
       job_t(beruf,gehalt,einstelldatum)
```

```
from angestellte_basis;
```

Beim Anlegen einer Objekt-View erfolgt die Referenzierung über die Namen der Konstruktormethoden. Daten können über Objekt-Views bzw. über die relationale Tabelle eingefügt (manipuliert) werden.

Objekt-Views mit Referenzen.

Gegeben sind 2 relationale Tabellen.

```
create table personen
(
 pers_id varchar2(3)
 constraint personen_PK primary key,
 name varchar2(10),
 strasse varchar2(15),
 ort varchar2(15),
 plz number
);
insert into personen values
('P01','Juergen','Kirchplatz 7','Bad Hersfeld',36251);
create table verabredungen
(
 pers_id varchar2(3),
 ruf_nr  varchar2(15),
 ruf_datum date,
 constraint verabredungen_PK
 primary key (pers_id, ruf_nr),
 constraint verabredungen_FK foreign key(pers_id)
 references personen(pers_id)
-- der Fremdschluessel definiert die Beziehung zwischen den beiden
-- Tabellen
);
```

Falls die Tabelle `personen` mit anderen Tabellen in Beziehung steht, kann über Objekt-Views eine Referenz zwischen beiden Tabellen angelegt werden. In diesem Fall benutzt Oracle die vorhandenen Primärschlüssel/Fremdschlüssel-Beziehungen zur Simulation eines OIDs der für REFs zwischen den Tabellen eingesetzt wird. Damit kann auf die Tabellen als relationale Tabellen oder als Objekte zugegriffen werden.

```
insert into personen values
('P02','Christian','Kirchplatz 5','Bad Hersfeld',36251);
insert into verabredungen values
('P01','06621-14163',trunc(sysdate) -1);
insert into verabredungen values
('P02','06621-14463',trunc(sysdate));
```

Aus objektorientierter Sicht referenzieren die Datensätze in `verabredungen` die Datensätze in `personen`. Ein abstrakter Datentyp, der dieselbe Struktur wie die Tabelle `personen` hat, kann dabei behilflich sein.

```
create or replace type personen_t as object
(
 pers_id varchar2(3),
 name varchar2(10),
 strasse varchar2(15),
 ort varchar2(15),
 plz number
);
/
```

Jetzt kann auf der Grundlage des Typs personen_t eine Objekt-View angelegt werden, OID-Werte werden den Datensätzen in personen zugewiesen.

```
create or replace view personen_ov of personen_t
with object identifier (pers_id)
-- damit koennen Zeilen in der Tabelle personen so adressiert werden, als
-- ob es sich um referenzierbare Zeilenobjekte einer Objekttabelle
handelt
as
select pers_id, name, strasse, ort, plz
from personen;
```

Spalten der Tabelle personen sind jetzt über personen_ov als Zeilenobjekte zugänglich, die für die Zeilen in personen_ov generierten OID-Werte werden deshalb pkOIDs genannt. Grundsätzlich können alle relationalen Tabellen als Zeilenobjekte zugänglich sein. Vorraussetzung ist allerdings, dass für diese Tabellen Objekt-Views angelegt sind.
Die Zeilen von verabredungen referenzieren Zeilen in person. Aus der relationalen Perspektive wird die Beziehung über Fremdschlüssel ermittelt, die von der Spalte verabredungen.pers_id auf personen.pers_id zeigt. Nachdem personen_ov angelegt wurde und die Zeilen in personen über OIDs zugänglich sind, müssen jetzt in verabredungen Referenzwerte angelegt werden, die personen referenzieren.

Generieren von Referenzen

```
create view verabredungen_ov as
select MAKE_REF(personen_ov,pers_id) pers_id, ruf_nr, ruf_datum
from verabredungen;
```

MAKE-REF legt Referenzen an (sog. pkREFs, da sie auf Primärschlüsseln basieren). Sobald die REFs eingerichtet sind, kann mit der DREF-Funktion aus verabredungen auf Daten in personen zugegriffen werden.

```
select DEREF(vov.pers_id)
from verabredungen_ov vov
where ruf_datum = trunc(sysdate);
```

3.2.2 Abfragen in Objektstrukturen

Objektinstanzen und ihre Attributwerte können aus Objekttabellen oder verschachtelt in Spalten abgefragt werden, wobei Attributwertedaten als Spaltenwerte in Zeilen oder als Strukturen von Objektinstanzen erscheinen.

3.2.2.1 Abfragen von Objekttabellen und Spalten

Eine Objekttabelle enthält ein oder mehrere Objektstrukturen, die ihrerseits ein oder mehrere Attribute umfassen. Falls auf der basis einer Objettabelle eine SELECT-Anweisung geschrieben wird, kann jede Instanz so behandelt werden, als ob es sich um eine Zeile in einer relationalen Tabelle handeln würde, und jedes Attribut in jeder Objektinstanz, als ob es eine Spalte in einer Zeile wäre.
Syntax einer solchen Abfrage:

```
SELECT spaltenliste
FROM tabellenliste
WHERE bedingung(en)
GROUP BY gruppierungskriterium
HAVING bedingung(en)
ORDER BY spalte(n)
```

In der `spaltenliste` können Attributnamen und in der `tabellenliste` Objekttabellennamen angegeben werden. Attributnamen können überall verwendet werden, wo eine Spalte erscheint.

3.2.2.2 Nützliche Funktionen für Objektinstanzen und Objekttabellen

Die Funktion VALUE

Diese Funktion wurde zur Rückgabe einer Objektinstanz entworfen. Es gilt:

- der Parameter für die Funktion VALUE() ist bei Verwendung in einer SQL-Anweisung ein Tabellenalias, der die Zuurrdnung zu einer Objektinstanz (oder Zeile) in einer Objekttabelle zur Verfügung stellt.
- der Rückgabewert der Funktion VALUE() ist eine Objektinstanz, deren Objekttyp dem Typ entspricht, der für die Definition der Objekttabelle verwendet wird.

- die Funktion REF

Diese Funktion wird zur Rückgabe einer Referenz auf eine Objektinstanz verwendet.

- die Funktion DREF

Diese Funktion gibt den Wert einer Objektinstanz über ein Verweisargument zurück. Das Argument muß ein Verweis auf die Objektinstanz sein.

3.2.2.3 Abfrage von Werten verschachtelter Objektinstanzen

3.2.2.4 Abfrage von Collection

Falls eine Objekttabelle auf einem Typ beruht, der eine Collection enthält (z.B. den Typ einer verschachtelten Tabelle), dann kann die Funktion TABLE() verwendet werden, um die verschachtelte Tabelle in eine Menge Zeilen und Spalten zu zerlegen. Die flache Struktur kann dann so behandelt werden, als würde es sich um eine herkömmliche, relationale Tabelle handeln, die mit hilfe von standardmäßigen SQL-Anweisungen verwendet werden kann.

3.3 Objektrelationale SQL Datenbankprogrammierung in Oracle

3.3.1 Anwendungsprogrammierung in PL/SQL (Object PL/SQL)

PL/SQL-Programme können mit den angegelegten abstrakten datentypen arbeiten.Der Verwendung eigener, benutzerdefinierter Datentypen steht nichts mehr im Wege. Das Ergebnis Ist eine Mischung aus SQL, prozeduraler Logik und objektorientierten Erweiterungen – eine Kombination, die Object-PL/SQL heißt.

3.3.2 Anwendungsprogrammierung mit Java

Im besten Fall ermöglicht die Anwendungsprogrammiersprache Java und entsprechende Datenbankzugriffsschnittstellen wie JDBC und SQLJ eine „1:1“-Abbildung zwischen Anwendungs- und Datenbankobjekten und erlauben insbesondere die transparente Verwendung von Datenbankmethoden in der Anwendung.

3.3.2.1 JDBC 2

3.3.2.1.1 Collection Typen

Der VARRAY-Typ wird in Java durch das Interface `Array` abgebildet.
Auch von einer NESTED TABLE kann über das Interface ARRAY gelesen werden.

3.3.2.1.2 CLOB, BLOB

Ein SQL-BLOB wird durch ein Java-Objekt mit der Schnittstelle Blob repräsentiert. Dieses Objekt enthält einen Lokator (logischer Zeiger) auf das SQL-BLOB. Das Gleiche gilt für die Darstellung eines SQL-CLOBs, das durch ein Java-Objekt mit der Schnittstelle `Clob` repräsentiert wird.

3.3.2.1.3 Benutzerdefinierter Typ

Zur Verarbeitung von SQL-Daten in Java müssen geeignete Abbildungen zwischen den unterschiedlichen Typsystemen von SQL und Java definiert sein. JDBC stellt Standardabbildungen zur Typkonvertierung zwischen SQL und Java zur Verfügung. Für jeden vordefinierten SQL-Datentyp gibt es eine repräsentative Java-Klasse bzw. einen Java-Typ.

3.3.2.1.3.1 Struct

Ein SQL-Tupel wird durch ein Java-Objekt mit der Schnittstelle Struct repräsentiert. Dieses Struch-Objekt enthält nicht die Felder (Attribute) des SQL-Tupels, sondern nur einen Lokator auf das SQL-Tupel.

3.3.2.1.3.2 Mit eigener Klasse

Eine Java-Klasse, die einen SQL-Datentyp repräsentieren soll, muß die Schnittstelle `SQLData` implementieren.

3.3.2.1.4 Referenz-Typ

Eie SQL-Referenz wird durch ein Java-Objekt mit der Schnittstelle `Ref` repräsentiert. Dieses Ref-Objekt stellt einen Lokator auf die SQL-Referenz dar.

3.3.2.2 SQLJ

Die statische Einbettung von SQL-Anweisungen in Java-Programmen wird mit Hilfe von **SQLJ** durchgeführt.

3.3.2.3 JPublisher

Oracle bietet über den Java Object Type Publisher einen sehr komfortablen Weg zur Umwandlung von Objekten und Pakete aus einer Datenbank. Dazu müssen diese Objekte als benutzerdefinierter Typ in der Datenbank erstellt werden.

Typen in Oracle

Typen mit JPublisher in Java-Objekte konvertieren

Oracle liefert das Kommandozeilenprogramm `jpub` mit seiner Client-Distribution. Dieses Tool wandelt deklarierte Benutzertypen aus der Datenbank in entsprechende Java-Klassen um.

4. XML-Datenbanken

4.1 Konzepte und Standardisierung

4.1.1 XML

Grundlage dokumentenzentrierter Beschreibungen bilden sog. Auszeichnungssprachen (**Markup Language**). Eine Auszeichnungssprache ist eine Sprache, die Regeln zur Auszeichnung von Textelementen bereitstellt. Beliebigen Textelementen können auf deklarative Weise Eigenschaften zugewiesen werden, wodurch deren Bedeutung (Semantik) hergeleitet werden kann.

Auszeichnungstyp	Auszeichnungssyntax
Starttags	<elementName[attributes]
Schlußtags	</elementName>
Leere-Element-Tags	<elementName[atributes]/>
Entity-Referenzen	&entityName; oder &paramaterEntiyName;
Zeichenreferenzen	&#decimalNumber; oder &#hexNumber;
Kommentar	<!-- comment→
Begrenzer des CDATA-Abschnitts	<![CDATA[cdata stuff]]>
Dokumenttyp-Deklaration	<!DOCTYPE name externel ID? [DTD Stuff]>
Verarbeitungsanweisungen	<?processorID data ?>
XML-Deklaration	<?xml version encoding standalone?>

Abb.: Übersicht zu XML möglichen Auszeichnungen

XML berücksichtigt sowohl bei den Auszeichnungen als auch bei Zeichendaten Groß- und Kleinschreibung

4.1.1.1 XML-Dokumente

Aufbau

Ein XML-Dokument besteht aus Prolog (optionale Präambel), optionalem Schema und einem Wurzelelement (Körper).
Der **Prolog** (optionale Präambel) kann umfassen

- in der ersten Zeile XML-Version und Zeichencodierung
  Bsp:. <?xml version = "´1.0" encoding="UTF-8"
- in der zweiten Zeile die DTD (Dokument-Typ-Deklaration)
  Bsp.: <!DOCTYPE adressbuch SYSTEM "adressbuch.dtd">
- in der dritten (optionalen) Zeile den XSL-Stylesheet, der zur Formatierung der XML-Datei verwendet wird.
  Bsp.: <?xml :stylesheet type = "text/xsl" href="demo.xsl"?>

Danach beginnt der **Körper** des Dokuments. Zuerst kommt das Wurzelelement. Es kann beliebig viele und beliebig tief geschachtelte Unterelemente beinhalten. Ein XML-Dokument darf nur eine Wurzel haben. Für jeden Starttag muß es auch

einen Schlußtag geben. Innere Elemente müssen immer geschlossen sein, bevor ein äusseres geschlossen werden kann. Der Tag, der zuerst geöffnet wurde, muß zuletzt geschlossen werden. Es handelt sich also um eine Baumstruktur.

Baumstruktur

Man kann die in XML-Dokumenten enthaltenen Informationen als Baumstruktur veranschaulichen:

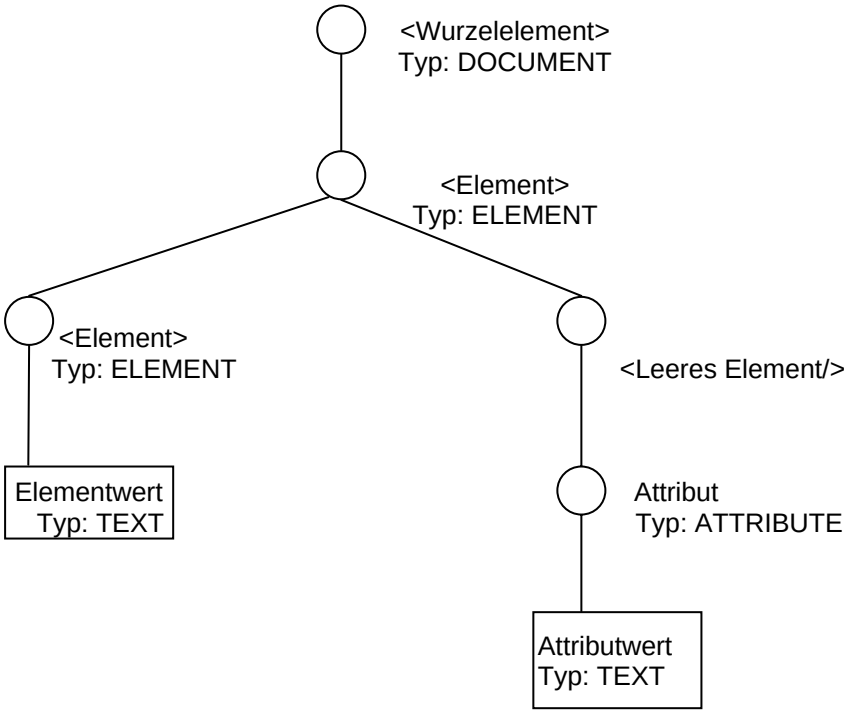


Abb.:

Wohlgeformte XML-Dokumente

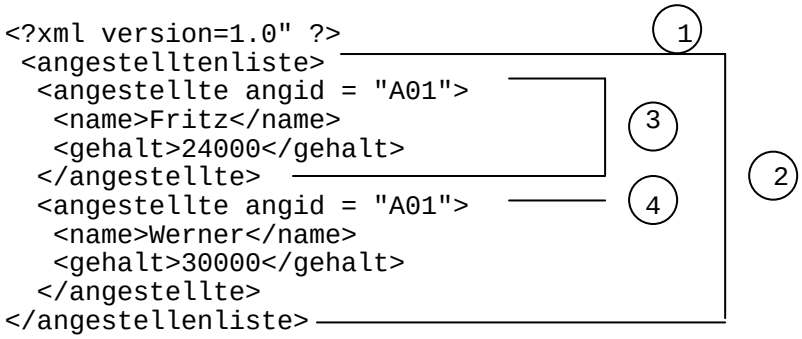
XML-Dokumente müssen bestimmte Syntaxregeln einhalten, z.B.:

Für jedes von Syntaxregeln definierte und verwendete öffnende Tag muß es ein schließendes Tag geben.

```
<?xml version=1.0" ?>
 <firstTag>
  <secondTag>
   <informationTag>information</informationtag>
  </secondTag>
 </firstTag>
```

Die Syntaxregeln sorgen für die hierarchische Integrität von Dokumenten.
Ein "wohlgeformtes XML-Dokument" muß folgende Syntaxregeln einhalten:

- Das Dokument muß mit der XML-Deklaration beginnen (1)
- Das Dokument muß über ein sog. Root-Element verfügen, das alle übrigen Elemente enthält. (2)
- Alle Elemente müssen über ein Start- und ein End-Tag verfügen, es sei denn, es handelt sich um leere Elemente (3)
- Leere Elemente müssen nur über ein Tag im Format `<name/>` verfügen
- Attributwerte müssen in Anführungszeichen gesetzt werden. (4)




```
<?xml version=1.0" ?>
 <angestelltenliste>
  <angestellte angid = "A01">
   <name>Fritz</name>
   <gehalt>24000</gehalt>
  </angestellte>
  <angestellte angid = "A01">
   <name>Werner</name>
   <gehalt>30000</gehalt>
  </angestellte>
</angestellenliste>
```

Bei folgenden Bestandteilen eines XML-Dokuments wird zwischen Groß- und Kleinschreibung unterschieden: Elementtypnamen, Attributsnamen, Attributwerte, alle allgemeinen Entity-Namen und Entity-Namen für Parameter.

Schema oder kein Schema

Ein XML-Dokument kann entweder ein Schema besitzen oder auch nicht. Falls einem Dokument ein Schema zugeordnet ist, muß dieses Schema auch eingehalten werden. Ein XML-Dokument mit einem Schema heißt valide oder gültig, falls es die im Schema festgelegten Anforderungen erfüllt.

4.1.1.2 Bestandteile der Sprache und deren DTD-Definition

Entity / Entities

In der XML-Terminologie werden Grundelelmente von XML-Dokumenten als ENTITY/ENTITIES bezeichnet, die entweder verabeiteten Text (parsed character data oder Rohtext (unparsed character data) enthalten . Der verarbeitete Text umfaßt sowohl den Dateninhalt als auch deklarative Textauszeichnungen, die die logische Struktur von Text beschreiben.
Syntax: `<!ENTITY name value>`
Allgemeine Entity-Deklaration zur Definition von Dokumentbestandteilen

```
<!ENTITY hansesailtermin “11.-14. August 2005“>
<!ENTITY hansesailteilnehmer “200 Traditionssegler und Museumsschiffe“>

<veranstaltung>
    Die Hansesail findet in diesem Jahr vom &hansesailtermin; statt.
        Es werden &hansesailteilnehmer; erwartet.
</veranstaltung>
```

Entityreferenz: *&Entityname;*

Vordefinierte Entities dienen zur Verwendung spezieller Zeichen:

- < für <
- > für >
- & für &
- ' für ´
- " für “
- ß für ß
- ä für ä
- Ä für Ä
- ö für ö
- Ö für Ö
- ü für ü
- Ü für Ü

Die Zeichen können nicht direkt im Text dargestellt werden, da sie eine besondere Bedeutung haben. Diese Entities kann jeder XML-Prozessor auswerten.

Character Entities dienen zur Definition von Sonderzeichen und Zahlen. Für Dezimalzahlen ist zu unterscheiden:

- Dezimalzahlen im Bereich 0..255
  - Verwendung der Zeichen der erweiterten ASCII-Menge (ISO 8859/1), auch Latin-1 genannt
- Dezimalzahlen im Bereich 256..65535
  - Verwendung der Zeichen des Unicode-Zeichensatzes (ISO 10646)
- Anstelle von Dezimalzahlen lassen sich Hexadezimalzahlen verwenden, diesen wird ein x vorangestellt

Beispiel: &#60 stellt das Zeichen < dar

Nichtgeparste Entities werden zur Einbindung anderer Dateiformate, z.B. Bilder und ähnliches verwendet.
Bsp.:

```
<!ELEMENT Hotel (#PCDATA)>
<!ATTLIST Hotel ansicht ENTITY #IMPLIED>
<!ENTITY ansicht_Huebner
  SYSTEM "ansicht_Huebner.gif" NDATA GIF>
<!NOTATION GIF SYSTEM 'gifviewer.exe‘>
<Hotel ansicht="ansicht_Huebner">
   Strand Hotel Hübner
</Hotel>
```

Parameter Entities braucht man zur Verwendung in DTDs. Auf eine Parameter-Entity bezieht man sich mit dem %-Zeichen[185].
Bsp.:

- `<!ENTITY % adressdef ´(ort,plz,strasse,nummer)´>`
- `<!ELEMENT adresse %adressdef;>`

- Einsatz zur Deklaration von mehrfach auftretenden Teilen einer DTD
- Ziel: Wiederverwendung, „Modularisierung“

Allgemeine externe Entities ersetzen den Entity-Namen durch den Inhalt einer Datei (Aufruf wie interne Entities). Die Datei wird über das Schlüsselwort SYSTEM gefolgt von URI (Uniform Ressource Identifier) referenziert.
Bsp.: Deklaration eines externen Entities

```
<!ENTITY legal SYSTEM ´/include/legalnotice.xml´>
```

Binäre externe Entities gibt es nur in Attributen vom Typ ENTITY/ENTITIES und umfassen beliebige Daten (keine XML-Syntax). Die angegebene Notation ermöglicht die Interpretation eines Dateityps durch ein Programm.
Bsp.:

```
<!NOTATION JPEG SYSTEM ´/usr/bin/xview´>
<!ELEMENT Haus (#PCDATA)>
<!ATTLIST Haus ansicht ENTITY #IMPLIED>
<!ENTITY ansicht_323 SYSTEM “323.jpg“ NDATA JPEG>

<Haus ansicht=´ansicht_323´> ... </Haus>
```

Elemente und Attribute

Ein **Element** kann bspw. umfassen: `<Name> Juergen </Name>`. Elemente

[185] Außerhalb der DTD hat das %-Zeichen keine Bedeutung

- sind XML-Grundbausteine
- umfassen Start- und Endtag (Metadaten), z.B. <Name> und </Name>
- enthalten dazwischen Daten (Elementinhalt) z.B. Juergen
- können auch leer sein: </Name>
- können auch Starttags mit Attributen enthalten

Auch Schachtelungen sind erlaubt

```
<Adresse>
 <Ort> Regensburg </Ort>
</Adresse>
```

Grundsätzlich sieht eine ELEMENT-Deklaration in einer DTD so aus: <!ELEMENT name content-model>. Als Inhaltsmodell (content-model) können beliebig terminale und nicht-terminale Elemente gewählt werden. Terminale Elemente werden durch den Typ #PCDATA beschrieben.
Elementinhalte können deshalb auch Elemente im Rahmen einer Element-schachtelung (neben Zeichenketten, Processing Instructions, Kommentare) sein. Eine **DTD** legt die Strukturierung (Schachtelung) und Benennung der Elemente bzw. Attribute fest.
Bsp.:

```
<!ELEMENT adresse (ort,plz,strasse,nummer,email)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT plz (#PCDATA)>
<!ELEMENT strasse (#PCDATA)>
<!ELEMENT nummer (#PCADATA)>
<!ELEMENT email (#PCDATA)>
```

#PCDATA bedeutet: Die Elemente sind atomar, dürfen also nur Parsed Character Data (Text) enthalten. Datentypen wie in SQL gibt es ansonsten nicht.
Die Strukturierung wird durch Ausdrücke beschrieben, die folgende Konstruktoren beinhalten können:

- Sequenz (A1,A2)
  Die Sequenz gibt die Reihenfolge der Elemente vor, hier muß A1 vor A2 kommen
- Alternative (A1 | A2)
  Hier muß entweder A1 oder A2 angegeben werden

Für die Wiederholung oder auch die Iteration gibt es mehrere Varianten. Sie sind an die Notation regulärer Ausdrücke angelehnt und unterscheiden sich in den Kardinali-tätsanforderungen:

- A* (beliebige Iteration)
  A kann beliebig oft (oder keinmal) angegeben werden
- A+ (nichtleere Iteration)
  A muß mindesetns einmal angegeben werden
- A? (optionales Element)
  Das Element A kann maximal einmal angegeben werden

Bsp.: `<!ELEMENT adresse(ort,plz,((strasse, nummer) | postfach))>`

Leere Elemente[186] werden in der DTD nach folgendem Muster angegeben:

[186] Hinweis: Auch leere Elemente können Attribute haben

<!ELEMENT typ EMPTY>

Syntax	Bedeutung
?	Kein oder einmaliges Auftreten
+	Einmaliges oder mehrfaches Auftreten
*	Kein oder mehrfaches Auftreten
(a|b)	Entweder a oder b, aber nicht beide
(a,b)	A gefolgt von b

Abb.: Frequenzindikatoren, die in der XML-DTD verwendet werden

Zusätzlich zu Elementen können auch **Attribute** definiert werden, z.B.:

```
<Geburtsdatum Kalender="gregorianisch"> 02.01.1967 </Geburtsdatum>
```

Ein Attributwert wird nach folgendem Muster angegeben:

```
<Elementname Attributname="Attributwert">Elementinhalt</Elementname>
```

Eine Attributliste definiert je Element mehrere Attribute. Dazu dient das folgende Muster:

```
<!ATTLIST Elementname Attributname Attributtyp Optionalität>
```

Neben dem Datentyp `CDATA` und dem Aufzählungstyp gibt es für Attribute die Möglichkeit, sie als `ID`- bzw. `IDREF/ IDREFS`-Attribute zu definieren:

- ID-Attribut: Verweis auf ein Element ähnlich zu einer OID in Objektmodellen, d.h. dokumentweit eindeutige Referenz
- IDREF-Attribut: Damit kann ein Element referenziert werden. Als Bedingung kann #IMPLIED[187] oder #REQUIRED[188] angegeben werden

Syntax	Bedeutung
#REQUIRED	Es wird kein Standardwert vorgegeben. In einem valid-Dokument muß hier ein Wert angegeben werden.
#IMPLIED	Das Attribut muß nicht unbedingt vorhanden sein
„default-wert“	Kein oder mehrfaches Auftreten
#FIXED “constant-value“	Es gibt einen Standardwert, dieser kann nicht überschrieben werden

Abb.: Standarddaten für Attribute

Bsp.: `<!ATTLIST Geburtsdatum Kalender CDATA #REQUIRED ‘gregorianisch‘>`

Insgesamt kann man folgende Attributtypen feststellen:

CDATA	Character Data: Zeichendaten, die einen String bilden
ID	Dokumentweit eindeutige Werte
IDREF/IDREFS	Referenz / Referenzen
ENTITY/ENTITIES	Referenz auf Entity bzw. Entities
NMTOKEN/NMTOKENS	Wort / Wörter
(wert1|wert2|…)	Wertaufzählung

[187] optionales Attribut
[188] obligatorisches Attribut

Notationen

Zu einer Notation gehört all das, was der XML-Prozessor nict verstehen und auswerten kann. Dabei sind sowohl binäre Daten denkbar als auch Text, der nicht für den XML-Prozessor bestimmt ist.
Syntax: `<!NOTATION name identifier “helper“>`

4.1.1.3 XML-Schema (Schemadefinitionssprachen)

Jedes Element in der XML Schema-Definition hat den Präfix `xsd`, der vorab mit dem XML Schema-Namensraum assoziiert wurde. Der Namenraum-Präfix (hier gemäß Konvention xsd) dient dazu, die Schemadefinitions-Tags wie `<xsd:attribute>` oder `<xsd:element>` „weltweit“ eindeutig zu benennen. Grundsätzlich werden bei XML-Schemata zwischen komplexen Datentypen und einfachen Datentypen unterschieden. Einfache Datentypen dürfen keine Elemente als Inhalt und keine Attribute besitzen.

Simple Types

Simple Type	Bsp.
string	
normalizedString	
token	
byte	-1, 126
unsignedByte	0, 126
base64Binary	
hexBinary	
integer	
positiveInteger	1, 126789
negativeInteger	-126789, -1
nonNegativeInteger	0, 1, 126789
nonPositiveInteger	-126789, -1, 0
int	-1, 126789675
unsignedInt	0, 1267896754
long	
unsignedLong	
short	-1, 12678
unsignedShort	0, 12678
decimal	-1.23, 0, 123.4, 1000.00
float	-INF, -1E4, -0.0
double	
boolean	true, false bzw. 1, 0
dateTime	
duration	
date	1999-05-31
gMonth	
gYear	1999
gYearMonth	1999-02
time	13.20:00.000
anyURI	
language	
ID	
IDREF	
IDREFS	
ENTITY	

NOTATION	
NMTOKEN	
NMTOKENS	

Abb.: Einfache Typen

Neue simple Datentypen können aus existierenden Datentypen gewonnen werden, in dem bspw. eine Einschränkung des Bereichs vorgenommen wird.

Unter- /0bergrenzen bei geordneten Wertebereichen:

- minInclusive, maxInclusive (Grenzwerte sind eingeschlossen)
- minExclusive, maxExclusive (Grenzwerte sind ausgeschlossen)

Längeneinschränkungen bei Zeichenketten

length. minLength, maxLength (genaue, minimale, maximale Länge)

Längeneinschränkung bei numerischen Datentypen

- totalDigits (max. Anzahl der Dezimalstellen)
- fractionDigits (Anzahl der Nachkommastellen)

Kardinalitätseinschränkung bei Listen

length. minLength, maxLength

Aufzählung der erlaubten Werte

enumeration

Bsp.:

```
<xsd:simpleType name = “Farben“>
<xsd:restriction base = “xsd:string”>
  <xsd:enumeration value = “rot”/>
  <xsd:enumeration value = “blau”/>
  <xsd:enumeartion value = “weiss”/>
</restriction>
</xsd:simpleType>
```

Vordefinierte Datentypen

- List: Liste von einfachen Datentypen (mehrfache atomare Typen, die durch Leerzeichen getrennt sind)

Bsp.:

```
<xsd:simpleType name = “Hobbies“>
  <xsd: list itemType = “xsd:string“/>
</xsd: simpleType>
```

- Vereinigung: Der Union-Typ enthält alle möglichen Werte eines simplen Typs oder eines List-Datentyps

Bsp.:

```
<xsd:simpleType name = “Fahrzeug“>
  <xsd:union memberType = “PKW LKW Motorrad Fahrrad …“/>
</xsd:simpleType>
```

anyType-Objekte können jeglichen Inhalt enthalten, d.h. einfache und komplexe Typen

Aus simplen Datentypen können neue komplexe Datentypen gewonnen werden.

Komplexe, benutzerdefinierte Datentypen

- Sequenz (sequence): Liste von Elementen mit festgelegter Reihenfolge

- Alternative: Auswahl eines der vorgegebenen Elemente

Bsp.:

```
<xsd:complexType name = “Fahrzeug“ abstract = “true“>
  <xsd:complexContent>
    <xsd:sequence>
      <xsd:element name = ”hersteller” type = ”xsd:string”/>
      <xsd:element name = ”typ” type = ”xsd:string”/>
    </xsd:sequence>
    <xsd:choice>
      <xsd:element ref = “ottomotor”/>
      <xsd:element ref = “dieselmotor”/>
    </xsd:choice>
  </xsd:complexContent>
```

- Subtypbildung durch Extension

Bsp.:

```
<xsd:complexType name = “FirmenFahrzeug”>
  <xsd:complexContent>
    <xsd:extension base = “Fahrzeug”>
      <xsd:sequence>
      <xsd:element name = ”Firma” type = ”xsd:string”/>
      <xsd:element name = ”Inspektion” type = ”xsd:date”/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
<xsd:complexType>
```

4.1.2 Zusammenspiel von XML und Datenbanken

XML-Dokumente können die ganze Bandbreite von Daten bis zu Volltextdokumenten einnehmen (dokumentzentriert, semistrukturiert, datenzentriert). Entsprechend eignen sich auch Speicherungsverfahren von der Dokumentenverarbeitung (Information Retrieval) bis zur Datenbanktechnologie. Es gibt keine optimale Lösung für alle Anwendungen, Dokumentcharakter spielt entscheidende Rolle

Native XML-DBS und (objekt-) relationale DBS mit erweiterter XML-Funktionalität

Zwei Wege werden derzeit zur XML-Unterstützung eingeschlagen:

- sog. **native** XML-Datenbanksysteme (XML-DBS)[189], die vollständig auf Speicherung und Verarbeitung von XML-Daten ausgerichtet sind. Der Einsatz nativer Systeme XML-DBS bietet sich an, wenn ausschließlich XML-Daten zu speichern und zu verarbeiten sind. Der Zugriff auf die Daten erfolgt dabei über eine XML-orientierte Schnittstelle (z.B. DOM oder XQuery).
- (objekt-) relationale DBS mit erweiterter XML-Funktionalität[190]. Die aktuelle Datenbank-Landschaft wird deutilich von (objekt-) relationalen DBS dominiert. Sehr viele Daten sind objektrelational gespeichert und eine hohe Anzahl von DB-Anwendungen basiert auf dem Zugriff von SQL. Die neueste Version der SQL-Norm, SQL:2003, sieht die Unterstützung von XML-Daten vor.

SQL-Basisdatentyp „XML“

Im Normteil „SQL/XML“ wird der SQL-Basisdatentyp „XML“ eingeführt, der mit einem Nullwert, einem XML-Dokument oder eine Folge von XML-Werten instanzierbar ist. Darüber wird es mögich, Tabellen anzulegen, die Spalten des Typs XML besitzen. Weiterhin definiert die Norm verschiedene Funktionen, die es erlauben, XML-Werte (Instanzen des XML-Datentyps) aus herkömmlichen SQL-Daten zu generieren.

Formen der Übergabe von XML-Werten

Die Nutzung von Tabellen mit XML-wertigen Spalten führt zur Frage, in welcher Art XML-Werte zwischen dem DBS und den Anwendungsprogrammen ausgetauscht werden sollen. Die SQL-Norm sieht hier zwingend die Übergabe in der Form von Zeichenketten vor. So werden XML-Werte beim Auslesen aus der DB (implizit oder explizit) mit Hilfe der SQL/XML Serialisierungsfunktion `XMLSERIALIZE` in Zeichenketten serialisiert und dann als `VARCHAR`-Werte oder `CLOB`-Werte an das Anwenderprogramm übergeben. Die entgegengesetzte Richtung (das Einbringen von XML-Werten in die DB) erfolgt analog. Das Anwendungsprogramm übergibt XML-Werte serialisiert als `VARCHAR`- bzw. `CLOB`-Werte an das DBS, dort werden diese dann (implizit oder explizit) über die SQL-

[189] z.B. Tamino XML Server von der Software AG

[190] sog. Hybriddatenbanken, z.B.: DB2 von IBM, Microsoft SQL-Server und Oracle

XML-Funktion `XMLPARSE` in Werte des Datentyps XML umgewandelt. Diese Form der Datenübergabe wird **textbasiert** genannt. Auch andere Übergabeformen sind denkbar: binär (z.B. als Baum im Falle eines XML-Dokuments), getrennte Übergabe von Strukturinformationen und eigentlichen Daten (z.B. numerische und boolesche Daten in natürlicher Form).

Zusammenspiel von relationalen Datenbanken und XML

Zwei Zielsetzungen sind beim Zusammenspiel von relationalen Datenbanken und XML zu unterscheiden.

1. Speicherung von XML-Dokumenten in relationalen DBS
2. Transformation relationaler Daten in XML-Dokumente

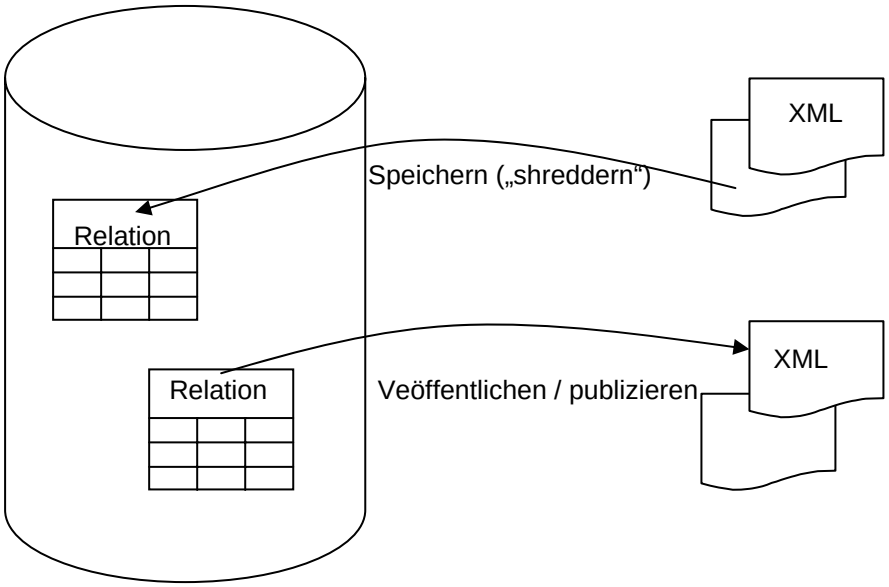


Abb.: Zusammenspiel relationaler DB und XML

Der neue Datentyp XML. Datenbank-Anbieter haben Unterstützung für Speicherung und Abfrage von XML-Dokumenten Speicherung und Abfrage in ihre relationalen DBS integriert: Für die Verwaltung von XML-Dokumenten gibt es in den relationalen DBS den Attributtyp `xml`, so daß man diesem Attribut ein XML-Dokument zuordnen kann, z.B.:

```
create table buecher (isbn varchar(20) primary key,
                                    Beschreibung xml)
```

Das Attrribut `beschreibung` vom Typ `xml` nimmt ein komplettes XML-Dokument auf.
Bsp.: Einfügeoprationen in die Tabelle `buecher`

```
insert into buecher values('3486273922',
          '<buch jahr="2006">
             <titel>Datenbanksysteme: Eine Einführung</titel>
             <autoren>
                <autor>
                   <vorname>Alfons</vorname>
                   <nachname>Kemper</nachname>
                </autor>
```

```
                    <autor>
                        <vorname>Andre</vorname>
                        <nachname>Eickler</nachname>
                    </autor>
                </autoren>
                <verlag>Oldenbourg Verlag</verlag>
            </buch>')

insert into buecher values('0136292399',
            '<buch jahr="1994">
                <titel>Object-oriented Data Management</titel>
                <autoren>
                    <autor>
                        <vorname>Alfons</vorname>
                        <nachname>Kemper</nachname>
                    </autor>
                    <autor>
                        <vorname>Andre</vorname>
                        <nachname>Eickler</nachname>
                    </autor>
                </autoren>
                <verlag>Prentice Hall</verlag>
            </buch>')
```

4.2 XML und Oracle

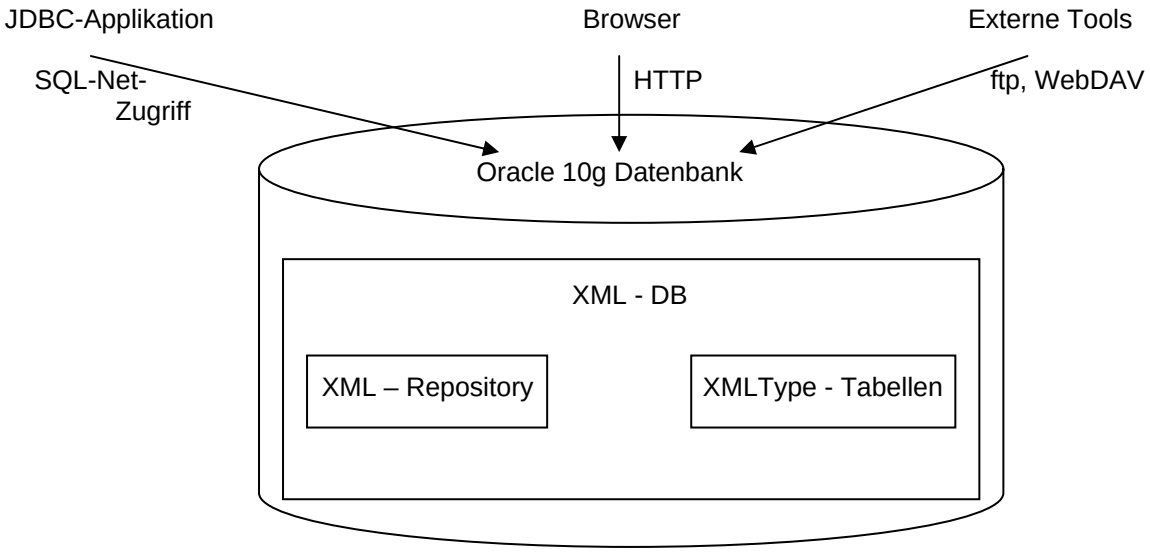


Abb.: XML-DB

Folgende Komponenten in Oracle9i bieten XML-Unterstützung, oder sind speziell für XML entwickelt worden:

- Speichern von XML-Daten in einer Tabelle vom Typ XMLType
- XML Dokumente als Daten speichern
- XML Dokumente als Dokumente speichern
- XML Parser und XSL Processors (Java, C, C++, PL/SQL)
- XML Class Generator (Java, C++)
- XML SQL Utilities für Java
- XSQL Servlet
- XML Transviewer Beans
- XDK-Tools (Software-Devoloper-Kits)

4.2 1 Speichern von XML-Daten in Oracle[191]

XML-Dokumente werden durch ihre Charakteristik in dokument- und datentechnische Dokumente unterteilt. Die Speicherungsform als CLOB erlaubt dokumentzentrisches Speichern. Die objektrelationale Speichervariante kommt dem datenzentrischen Typ entgegen. Damit die DB zusätzliche XML-Zugriffsweisen bereitstellt, steht der Datentyp XMLTYPE zur Verfügung. Bei einfacher Angabe von XMLTYPE wird dadurch eine dokumentzentrische Speicherung als CLOB möglich.

XMLType

Mit XMLType kann man XML-Daten in einer Oracle Datenbank speichern und abfragen. Als Typ besitzt XMLType Member-Funktionen für Zugriff, Extrahieren

[191] vgl. Ulrike Schwenn: XML in der Oracle Datenbank „relational and beyond“, http://doesen0.informatik.uni-leipzig.de/proceedings/slides/btw2003_ind_schwinn.pdf Zugriff: April 2007

und Abfragen von XML-Daten mit einer Klasse von Operatoren, die man XPath-Ausdrücke nennt.

Einsatz von standardkonformen annotated XML-Schemas

Zur Beeinflussung objektrelationaler Speicherung sind Beschreibungen für die XML-Strukturen nötig. Dies geschieht durch den Einsatz von standardkonformen annotated XML-Schemas, die die Oracle-spezifischen Beschreibungen als Attribute in den Schema-Annotations mit liefert. Ein **annotaded XML-Schema**

- ist W3C standardkonform[192].
- ist ein XML-Schema mit Zusatzinformation (hier: Oracle spez. Objektnamen, Speicherung, etc.).
- bei der Schema-Überprüfung ignoriert der Parser gemäß W3C alle unbekannten Namespaces. Datenbank-unabhängige Parser beachten nur das allgemeine XML-Schema
- kann Annotations für verschiedene Datenbanken enthalten, da die jeweiligen Namespaces abgegrenzt werden.

Bsp.:

4.2.2 Funktionsvorrat zur Unterstüzung von XML-Applikationen

Query-Rewrite

In einer objektrelationalen Datenbank ist es natürlich möglich, über SQL-Zugriffe Daten abzufragen oder zu verändern. Dabei ist jedoch die Integration der XPATH-Funktionalität (XML Path Language) in den SQL-Abfragen wichtig, um XML-Anfragen standardgemäß implementieren zu können. Bei objektrelationaler Speicherung besteht die Möglichkeit, den Zugriff direkt auf Objekte abzubilden (**Query Rewrite**).

SQL/XML

Zur Erzeugung komplexer XML-Strukturen unterstützt Oracle Abfragen im SQL-Stil, die unter dem Namen SQL/XML bekannt sind. SQL/XML wird durch die SQLX-Gruppe standardisiert. SQL/XML ist eine Erweiterung zu SQL und führt neue Funktionen und Operatoren ein. Mit Funktionen wie z.B. XMLElement, XMLAttributes in SQL-Abfragen kann der Programmierer XML-Dokumente generieren, die beliebig mit anderen relationalen oder XML-Tabellen kombiniert werden können.

Repository

Abgesehen von datenbankspezifischen Zugriffen durch SQL-Kommandos stehen stehen für den Zugriff von beliebigen Client-PCs (ohne zusätzliche Information) Protokoll-Schnittstellen wie HTTP, WEBDAV (Web Distributed Authoring and

[192] In solchen XML-Schemas werden inder Regel mindestens 2 Namensräume angegeben – einen für W3C (http://www.w3c.org/2001/XMLSchema) und einen für die Oracle-Charakteristiken (http://wmlns.oracle.com/xdb)

Versioning) und FTP zur Verfügung. Dies ermöglicht dieErweiterung des DB-Modells um Funktionen und Verzeichnisse, Versionierung und ACLs (Access Control Lists), die in einem Repository in der DB liegen. Dieses Repository, das automatisch in der Datenbank zur Verfügung steht, bildet die hierarchischen Verzeichnisstrukturen auf XML-Resourcen wie XML-Dateien oder Schemas ab.
Der Anwender kann so bspw. Oracle-XML-Datenbankverzeichnisse öffnen oder Web-Verzeichnisse im Internet-Explorer nutzen.

XSQL-Servlet

Es ermöglicht auf Tabellendaten über URLs zuzugreifen. Nur durch Angabe einer URL im Browser können somit ohne zusätzliche SQL-Programmierung, relationale Daten angezeigt werden. Das Servlet unterstützt das Generieren von XML- und Nicht-XML-Inhalten wie auch das Transformieren der Resultate durch Stylesheets[193].

4.2.3 XML-SQL Utility

Mit der XML-SQL Utility (**XSU**) von Oracle können die Ergebnisse einer SQL-Abfrage in XML konvertiert werden. XSU unterstützt das dynamische Generieren von DTDs und das Einfügen von XML in Datenbanktabellen.

[193] erledigt eine Stylesheet-Prozessor, der direkt in der DB liegt.

Empfohlene Literatur

Date, C.J.: An Intoduction to Database Systems, Volume I, Fifth Edition, Addison-Wesley, Reading Massachusetts,1990
Date, C.J.: An Introduction to Database Systems, Volume II, Addison Wesley, Reading Massachussets, 1985
Gardarin, G. / Valduriez, P.: Relational Databases and Knowledge Bases, Addison-Wesley, Reading Massachusetts, 1989
Heuer, A.: Objektorientierte Datenbanken, Addison-Wesley, Bonn ..., 1992
Ullman, J. D.: Database and Knowledge-Base Systems, Volume I, Computer Science Press, Rockville, 1988
Vossen, G.: Data Models, Database Languages and Database Mangement Systems,Addison-Wesley, Wokingham, 1990
Wedekind, H.: Datenbanksysteme I, BI Wissenschaftsverlag, Mannheim ..., 1974
Froese, Jürgen / Moazzami, Mahmoud / Rautenstrauch, Claus / Welter, Heinrich: Effiziente Systementwicklung mit ORACLE7, Addison-Wesley, Bonn ..., 1994
Herrman, Uwe / Lenz, Dierk / Unbescheid, Günther: Oracle 7.3, Addison-Wesley, Bonn ... , 1997
Ault, Michael R.: Das Oracle8-Handbuch, Thompson Publishing Company, Bonn ..., 1998
Türker, Can /Saake, Gunter: Objektrelationale Datenbanken, dpunkt.verlag, Heidelberg, 2006
Loney, Kevin : ORACLE DATABASE 10g Die umfassende Referenz, Hanser Verlag, München Wien 2005
Skulschuss, Marco / Michaelis, Samuel / Wieserstein, Marcus: Oracle 10g Programmierhandbuch, Galileo Press GmbH, Bonn 2004

Übungen

Übungsblatt-Nr	Überschrift	Adresse[194]
1. Übungsblatt	Datenbankentwurf zum Speichern von Daten aus dem Personal-wesen	http://fbim.fh-regensburg.de/~saj39122/DB/index.html
2. Übungsblatt	ERM-Diagramm füpr die DB zum Personalwesen	
3. Übungsblatt	Entwurf einer relationalen DB für das Personalwesen	
4. Übungsblatt	Implementierung einer relationalen DB für das Personal-wesen	
5. Übungsblatt	Abfragen an die relationale DB für das Personalwesen	
6. Übungsblatt	Mehrbenutzerbetrieb mit konkurrierenden Zugriffen	
7. Übungsblatt	Erstellen von Tablellen bzw. Sichten (views) aus Tabellen	
8. Übungsblatt	Hierarchische Beziehungen in relationalen DB	
9. Übungsblatt	Integritätssicherheitsvorkehrungen in Oracle-DB	
10. Übungsblatt	Embedded SQL	
11. Übungsblatt	Embedded SQL mit SQLJ	
12. Übungsblatt	JDBC	
13. Übungsblatt	Die objektorientierte relationale DB in Oracle	
14. Übungsblatt	PL/SQL	
15. Übungsblatt	PHP	
16. Übungsblatt	XML in Oracle 9i bzw. Oracle 10g	

[194] http://fbim.fh-regensburg.de/~saj39122/DB/index.html

Aufgabenblätter

1. Aufgabenblatt	Entwurf, Implementierung einer Erzeugnis-DB	Adresse[195]
2. Aufgabenblatt	Entwurf, Implementierung, Lieferant-Teil-Projekt -DB	
3. Aufgabenblatt	Embedded SQL	
4. Aufgabenblatt	Klausur	
5. Aufgabenblatt	Entity-Relationship-Diagramme	
6. Aufgabenblatt	Normalisierung von Tabellen	
7. Aufgabenblatt	UML-Diagramme und relationale / objektrelationale Darstellung	
8. Aufgabenblatt	Formate für Datumsangaben	
9. Aufgabenblatt	SQL-Queries	
10. Aufgabenblatt	JDBC	
11. Aufgabenblatt	Objekt-relationale-DB	
12. Aufgabenblatt	Nordwind-DB	
13. Aufgabenblatt	XML-DB	

[195] http://fbim.fh-regensburg.de/~saj39122/DB/index.html